# A Modelling and Simulation Tool for DNA Strand Displacement Systems

Shiting Long

**School of Science**

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 15.5.2020

**Supervisor**

Professor Pekka Orponen
Professor Vitaly Skachek

**Advisor**

Dr. Vinay Gautam

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

**Author** Shiting Long

**Title** A Modelling and Simulation Tool for DNA Strand Displacement Systems

**Degree programme** Computer Science

**Major** Security and Cloud Computing **Code of major** SCI3084

**Supervisor** Professor Pekka Orponen
Professor Vitaly Skachek

**Advisor** Dr. Vinay Gautam

**Date** 15.5.2020 **Number of pages** 63+7 **Language** English

**Abstract**

DNA is the hereditary material in almost all organisms, and the sequence of its monomers efficiently conveys essential biological information. Although DNA is well known for its biological functions, the unique material properties of DNA also motivate scientists to design and manufacture DNA complexes for technological purposes. This research field is termed *DNA nanotechnology*, and it aims to construct arbitrary biomolecular structures using DNA molecules as building blocks.

DNA nanotechnology initially focused on programmable static structures, but it has further inspired the designs of engineering systems with dynamic properties such as logic circuits and catalytic systems. This dynamic variant of DNA nanotechnology is enabled by the *DNA strand displacement* (DSD) mechanism. The design of a DSD system involves discreetly designed initial species that can execute expected sequential reactions. However, such task is hard to be accomplished by hand as the complete reaction network of a large-scaled DSD system can be intractable.

In this thesis, we study the problem of modelling DSD systems, i.e., enumerating combinatorially the full space of molecular complexes reachable from the initial species and transferring the resulting chemical reaction network to a simulation engine. We present a rule-based modelling pipeline RuleDSD for generating and simulating reaction networks of DSD systems. RuleDSD is implemented as a software package DSDPy, a tool that automatically generates a complete reaction network for a described DSD system and integrates with the PySB framework for further simulations using the BioNetGen engine. The reaction networks produced by DSDPy show that it is suitable for modelling various DSD systems from existing literature.

**Keywords** DNA Nanotechnology, DNA Strand Displacement, Rule-based Modelling, DSD Modelling and Simulation, PySB, BioNetGen

# Preface

Firstly, I would like to thank my main supervisor Professor Pekka Orponen and my advisor Dr. Vinay Gautam for offering this fascinating project presented in my thesis and providing me guidance and patience all along. I would also like to thank my second supervisor Professor Vitaly Skachek for encouraging me to pursue this thesis topic in the beginning and supporting me whenever I need his help.

I would further extend my gratitude to the professors and coordinators that I have met in the SECCLO programme, especially Professor Tuomas Aura, Dr. Ago-Erik Riet, Laura Mursu and Eija Kujanpää. The two years I spent in this programme are exceptional and unforgettable, I will bear this experience in mind as I continue my journey of life.

Finally, I would like to express my love for my family and friends, without them I would not be the person I am today.

Espoo, 15.5.2020

Shiting Long

# Contents

# 1 Introduction

Deoxyribonucleic acid (DNA) is a molecule found in all known living organisms that contains biological instructions for their development, survival and reproduction. DNA usually exists as a winding two-stranded structure, which is described as a *double helix*. The double helix is mainly stabilized by hydrogen bonds and base-stacking interactions that come from the complementary nucleobases pairing (A-T, G-C) between the two strands, which also induces the hybridisation between two DNA strands strictly following the base pairing rule.

The nanoscale biomolecular infrastructure of DNA wraps up nature's most complex secrets, and thus motivates scientists to uncover the hidden information. Although the biological properties of DNA have drawn most of the attention, the material properties of DNA also attract researchers to nano-fabricate DNA complexes by investigating and manipulating the interactions between DNA molecules.

In nanotechnology, self-assembly is a bottom-up approach which automatically generates extensive structures by preparing nanostructures in a defined arrangement. DNA is a remarkable candidate material for self-assembly because of the base pairing rule it obeys, as it makes the interactions between DNA molecules predictable. Seeman first recognised this capability of DNA molecules serving as non-biological building blocks for programmable designs of multilayered DNA complexes in 1982 [38]. The study of this field was later termed *DNA nanotechnology*, and it was named *structural DNA nanotechnology* when the finally assembled structures are static [39]. Over the following years, a cube [4] and a truncated octahedron [43] made of DNA molecules were synthesised using Seeman's methodology. As the field broadens, more advanced techniques have been introduced in DNA nanotechnology to increase the complexity of DNA structures, such as DNA origami [36] and DNA tiling [50].

In addition to enabling the construction of complex DNA structures, DNA nanotechnology is becoming attractive in designing synthetic biochemical systems with dynamic properties [55]. Dynamic DNA systems are usually based on the DNA strand displacement (DSD) mechanism [53, 56], a process in which a strand that hybridises partially to another strand displaces one or more pre-hybridised strands. It can be initiated by a single-stranded short domain, referred to as *toehold*. The feature of toehold-mediated DNA strand displacement allows the design of DNA systems with a focus on the reaction networks instead of the finally assembled static products. A wide variety of DNA strand displacement systems have been designed over the past two decades, such as DNA tweezers [53], DNA walkers [42, 44], DNA circuits [37, 33] and enzyme-free catalytic systems [56, 57].

## 1.1 Problem Statement

DSD systems use the DNA strand displacement mechanism as the main process to perform computations. This requires the designer of the system to plan very carefully so that the strand displacement reactions can happen in the right place and at the right time. Since the DSD mechanism usually involves a list of consecutive reactions and DSD systems may contain multiple sets of complexes associated with these reactions, a direct hard-coded design of a DSD system can be exhausting. Moreover, failing a laboratory experiment repeatedly can be expensive. In addition, the intricate nature of DNA molecules opens up numerous possibilities when DNA systems are scaled up. *Crosstalk*, i.e., DNA molecules undergoing a reaction unexpectedly affecting other reactions, is common in DSD systems. Therefore, an automated software for designing and simulating DSD systems prior to experimentation is in demand as the complexity of DSD systems increase.

Intuitively, one can model the DNA strand displacement mechanism by formulating the available reactions in this mechanism. Such reactions can be grouped into a small number of categories that follow specific rules. Hence, these rules can be applied on a given set of DNA molecules, where each application of a rule on one or more DNA molecules represents a reaction. The outcome thus consists of a set of all possible reactions and a renewed set of DNA molecules reachable from the given set. Starting from a set of initial DNA molecules, repeated applications of these rules to the current set of DNA molecules until there is no new DNA molecule produces the full reaction network of the system [31].

The modelling insight of DSD systems has motivated the development of automated design tools over the years. Visual DSD [25] is a tool for prototyping and analysis of DSD systems using the DSD programming language and its compiler [31, 24]. Visual DSD provides a programming interface through which a program adopting the DSD language syntax can describe a DSD system. Thus, the corresponding DSD language compiler can derive the set of all possible reactions based on a semantic analysis with the reduction rules it defines. Then the produced reaction network can be prompted for simulation. Other tools such as KinDA [1] and DyNAMiC Workbench [17] also work as modelling tools for DSD systems, though both use an alternative enumeration method to condense the reaction network by eliminating fast transients [16].

Current DSD modelling tools face mainly two challenges. The first one is that the enumerations over renewed sets may be computationally infeasible. The second one is that DNA molecules involving complex secondary structures such as loops [51], multiple branches [22] and pseudoknots [3] that cannot be expressed by previously developed tools have emerged in DSD system designs. To address the first challenge, researchers have proposed enumerators that cut out avoidable reactions with the sacrifice of full networks [16]. To address the second challenge, new methods have been proposed as amendments to the existing tools [46, 30]. At the same time, rule-based modelling languages such as BioNetGen [18] and Kappa [8] have successfully solved

similar challenges for modelling secondary structures in general biochemical systems. In fact, a new implementation for Visual DSD was inspired by Kappa [30]. Although DSD systems can be expressed in rule-based models using graphical structures to represent DNA molecules, the DSD language cannot be encoded in Kappa [30].

The attempt to describe DSD systems by rule-based models has inspired us to develop a tool that directly uses a rule-based modelling approach on graphical structures. The tool combinatorially enumerates the full space of reachable DNA complexes from a set of given initial DNA complexes, and then exports the resulting chemical reaction network to a simulation engine. This method nevertheless saves the time required to study the syntax of new programming languages, such as the DSD programming language used in Visual DSD and BioNetGen modelling language (BNGL). The architectural design of our tool constitutes the RuleDSD pipeline [13], a rule-based modelling and simulation software design for DSD systems. The methods comprising the RuleDSD pipeline are implemented as DSDPy, a tool that simplifies modelling and simulation of custom-made DSD systems. DSDPy also contains a simple interactive interface, which allows users to pause, resume and prematurely stop during the enumeration process to provide intermediate outputs for incremental analyses.

## 1.2 Structure of the Thesis

The remaining sections are organised as follows. Section 2 contains an introduction to the basic concepts we discuss in this thesis, including DNA structure and domain-level DNA strand displacement systems. In addition, we provide an overview of the rule-based modelling approach and the BioNetGen software that implements this approach for modelling biomolecular systems.

Section 3 provides a sketch of the RuleDSD pipeline and the definitions of critical concepts used in it. Further, a brief description of the four components constituting the RuleDSD pipeline is presented. Following the sketch, we give the details of important graphical structures that represent DNA molecules and prove the existence of a unique textual representation for each DNA molecule in the DSD system.

Section 4 explains the technical details of the RuleDSD implementation, which include the procedures and algorithms underlying the four components in the RuleDSD pipeline. We highlight the definitions of each rule defined for RuleDSD as they are the cornerstones in the rule-based modelling approach.

Section 5 presents the DSDPy tool, an implementation of the RuleDSD pipeline in Python. Further, this section provides an overview of its graphical user interface and gives a walkthrough of the tool for users. Finally, an analysis of two example DSD systems studied using DSDPy is discussed in comparison to the modelling of the same DSD systems using other state-of-the-art tools.

Section 6 serves as a conclusion to this thesis. We review the major outcomes of this thesis and discuss future possibilities for the RuleDSD pipeline and the DSDPy tool.

# 2 Background

In this section, we start with an introduction to the structure of DNA, then we illustrate the DSD mechanism, focusing on domain-level DSD systems. We also discuss the rule-based modelling approach used for biochemical systems in the end, as it inspired us to design a user-friendly tool for modelling DSD systems which saves the users from acquiring the knowledge of new programming languages such as BNGL and the DSD programming language used in Visual DSD.

## 2.1 DNA

DNA molecules have multiple conformations, at least three of them are confirmed to exist in nature: A-DNA, B-DNA and Z-DNA. The three conformations differ in their geometry and dimensions. B-DNA, the double helical structure that was first described by Watson and Crick in 1953 [48], is believed to be the most common conformation in cells [35]. The model of B-DNA as illustrated in Figure 1 is widely accepted as the structure for DNA molecules.

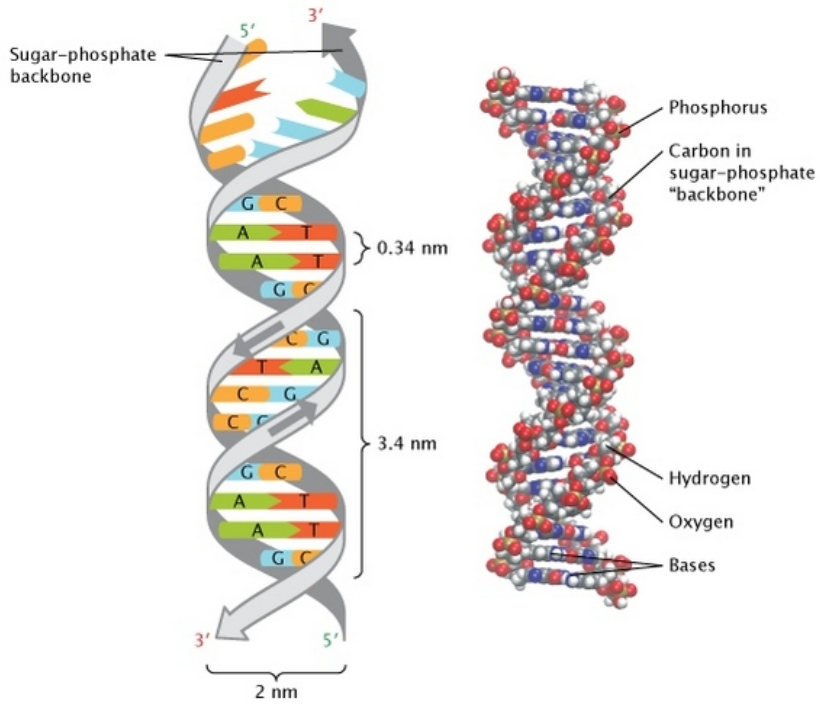


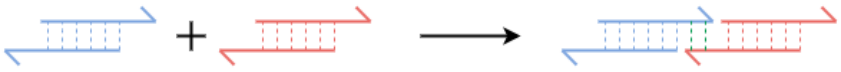

Figure 1: The double helical structure of DNA elucidated by Watson and Crick. Image reprinted from [32].

DNA has a double-stranded structure, where each strand known as a *polynucleotide* is composed of multiple nucleotides. A nucleotide has three major components: a

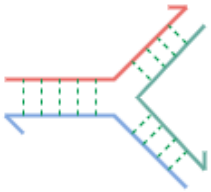
nucleobase, a five-carbon sugar, and a phosphate group. The exact nucleobase on a nucleotide differentiates it from others. DNA contains four types of nucleobases: cytosine (C), guanine (G), adenine (A) and thymine (T). The nucleotides are linked together as a strand by bonds between the sugar of a nucleotide and the phosphate of the next, and thus form a *sugar-phosphate backbone*. The two strands are hybridised by hydrogen bonds between pairs of complementary nucleobases (i.e., C with G and A with T).

The primary features of the Watson and Crick DNA model are as follows. Firstly, each DNA strand has an inherent orientation because the organisation of a sugar-phosphate backbone distinguishes two ends of a strand, known as the 5' end and the 3' end. Secondly, the two DNA strands in the model have the complementary sequences of nucleotides and wind around in opposite directions to each other, resulting in an *antiparallel* structure. Lastly, DNA follows a strict base pairing rule such that A nucleobases are always paired with Ts and Cs with Gs.

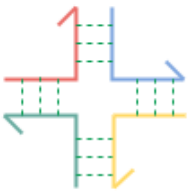
Three characteristics make DNA an ideal material for structural self-assembly approach: hybridisation, stable branched DNA (see Figure 2), and convenient synthesis of designed sequences [40]. The hybridisation of DNA not only connects two DNA strands to form a double helix but can also combine two double helical DNA molecules as shown in (a). Moreover, triply branched replication forks [49] and four-arm branched Holliday junctions [21] are known DNA structures as well (see (b) and (c)). These structures together with hybridisation provide rich possibilities in designing topologically structured DNA molecules and simulating their reactions. The capability of synthesising DNA molecules comprising arbitrary sequences allows researchers to make these possibilities come true in reality.



(a) Hybridisation of two molecules using single-stranded sticky ends.



(b) Illustration of a triply branched DNA.



(c) Illustration of a Holliday junction.

Figure 2: Hybridisation of DNA and stable branched DNA structures. Here, solid lines represent DNA strands in the 5' end to the 3' end (arrowhead) orientation, and dotted lines represent complementary base-pairing between two antiparallel DNA strands.

## 2.2 Domain-level DNA Strand Displacement System

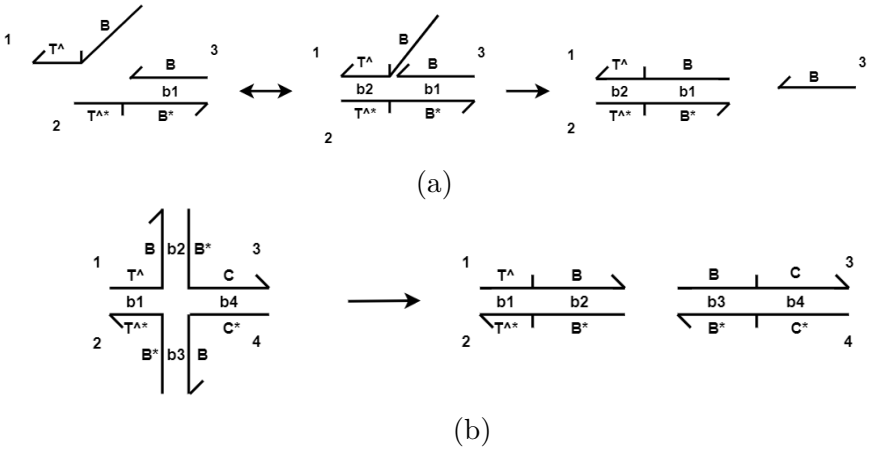
In DNA nanotechnology, DNA systems are usually studied at two levels: sequence-level and domain-level. Sequence-level DNA systems focus on the designs and interactions of DNA molecules at the level of nucleotides. Instead of taking nucleotides as the smallest units, domain-level DNA systems use contiguous nucleotide sequences of certain lengths, which are termed *domains*, as the smallest units. In other words, strands in domain-level DNA systems are conceptually subdivided into functional domains [56].

A DNA domain is usually assigned a specific name to separate the sequence it represents from the other sequences. The complementary domain of a certain domain nevertheless represents a complementary sequence so that the two complementary domains can be paired. Recall that a DNA strand has an inherent orientation: we consider the nucleotide sequence as oriented from the 5' end to the 3' end throughout this thesis. Therefore, given a domain with the sequence 'ACAG', its complementary domain would have sequence 'CTGT' because the pairing should be antiparallel. In this way, the idea of base pairing rules is inherited by the domain-level DNA systems.

Notice that there is a possibility that the sequence represented by some domain might be the same sequence that its complementary domain represents, e.g., a sequence 'CCGG'. However, we do not allow a domain to pair with itself as it is against the idea of strict pairing between two complementary domains. Hence, we do not consider the possibility that two complementary domains represent the same sequence in domain-level DNA systems in the scope of this thesis.

As sequence-level DNA systems provide comprehensive details, the information they contain can be redundant. Domain-level DNA systems preserve the principles of DNA interactions and simplify the representations of DNA molecules, which makes them optimal for modelling dynamic DNA systems. Hence, we model domain-level DNA systems in RuleDSD. Further discussions of DNA systems are therefore in domain-level description.

We give the description of domains as follows. A domain name is conventionally a combination of letters and/or numbers. We append a '$*$' symbol to a domain name to represent the complementary domain. For example, a domain $A^*$ is the complementary domain of the domain $A$. Domains in DSD systems have long sequences of nucleotides, e.g., 25-30 in length by the standards of Visual DSD [30]. There are short domains with length 4-10 as well [31], which we term *toehold* domains. We append a '^' symbol to a domain name to indicate that it is a toehold domain. Toehold domains are short enough to bind swiftly without additional binding [56, 52]. On the other hand, toehold type bonds, i.e, bonds constituting of toehold domains, are also actively available for an unbinding reaction when no adjacent bonds are formed.




(a)

(b)

Figure 3: Examples of DNA strand displacement system. We use a bond name, e.g., $b1$ to represent a bond between two domains.

Figure 3 (a) shows a simple DNA strand displacement system. In the initial state of the system, there is a double-stranded DNA molecule and a DNA strand. Since toehold domain $T\hat{}$in strand 1 has a complementary domain $T\hat{}^*$ in strand 2, a binding reaction is feasible. After the bond between the toehold domains is formed, strand 1 has two available options: 1) unbind with strand 2, and 2) initiate a reaction called *three-way branch migration* [56], through which the following domain in strand 1 replaces the same domain in strand 3 and binds with the complementary domain in strand 2. This process then releases strand 3 from the duplex.

Three-way branch migration is not the only type of reaction that results in strand exchange. In fact, a similar reaction happens where there are four strands concerned as illustrated in Figure 3 (b). If there is a Holliday junction as depicted, the bonds between two pairs of complementary domains can be switched, resulting in two double-stranded molecules. This reaction is called *four-way branch migration* [7].

The branch migration reactions convey a dynamic attribute of DSD systems. This simple and robust DSD mechanism makes it possible to design various dynamic systems based on DNA molecules. For instance, a two-input AND gate represented by a DSD system was proposed in [37]. As illustrated in Figure 4, the output strand is only produced when the two inputs are in presence with the reactant. Other DNA logic gates such as OR and NOT gates have also been constructed [37]. These results demonstrate the possibility that large and reliable circuits can be built up by DNA molecules, which means that the DSD systems can be powerful tools for programming biology [55]. Moreover, it was shown that any physically realistic abstract chemical reaction network (CRN) can be implemented by a DSD system of approximately equivalent behaviour [45], and any arbitrary linear input/output system can be constructed by DNA molecules [28].

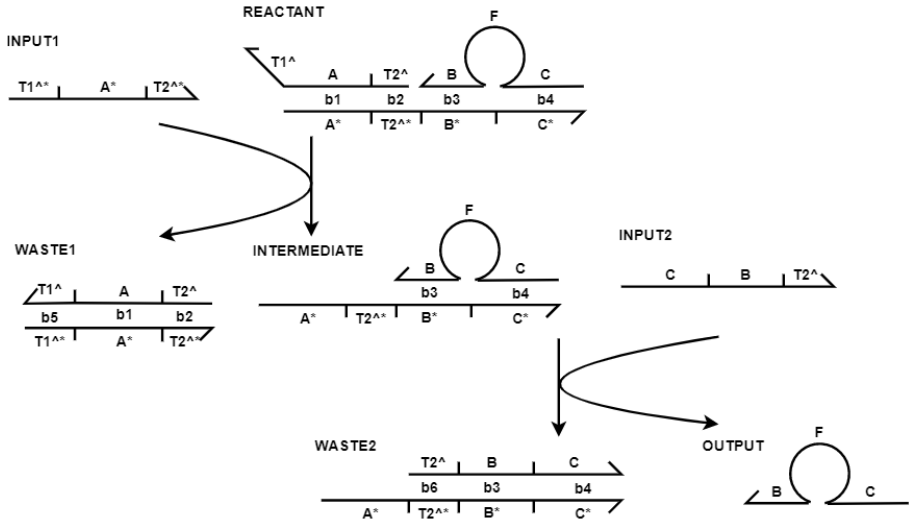



Figure 4: A two-input AND gate DSD system reported in [37].

It has been widely acknowledged that there are four types of reactions in DSD systems: binding, unbinding, three-way branch migration and four-way branch migration (illustrated in Figure 5). In a binding reaction, a domain binds with its complementary domain, resulting in a duplex. An unbinding reaction performs contrarily to binding as a duplex dissociates. These two reactions are the fundamental reactions in DSD systems with which branch migration reactions can be associated and design with the help of intricate, complex DNA molecular structures.

These types fit naturally with the idea of rules in a rule-based modelling approach which we discuss in the following section. Thus, we call them *DSD rules* in this thesis. The DSD rules are explained in detail in Section 4.1.

Particularly, the design of the toehold domains plays an important role in DSD systems as they are frequently considered actively available for binding and unbinding, and thus mediate branch migrations. Toehold domains can also be inactive if they are buried within bonded long domains [37] or hidden inside a loop [9, 51].

So far we have explained domain-level DSD systems and the set of DSD reactions, we need to further consider the reaction rates in order to make reliable predictions about the kinetics of the reaction network. However, the kinetics of real physical systems will be affected by external parameters such as temperature and salt concentrations, and these variables are not included in our modelling approach. To illustrate the calculation of reaction rates, we use the formula given by Grun et al. [16] where the rates are approximated at 25°C and with 10 mM $Mg^{+}$. Let $\rho(r)$ be the reaction rate of reaction $r(A, B)$, where $A$ is the reactant and $B$ is the product. Note that $A$ and $B$ may contain more than one species, i.e., $A = a_1, a_2, ...$ and $B = b_1, b_2, ....$ We

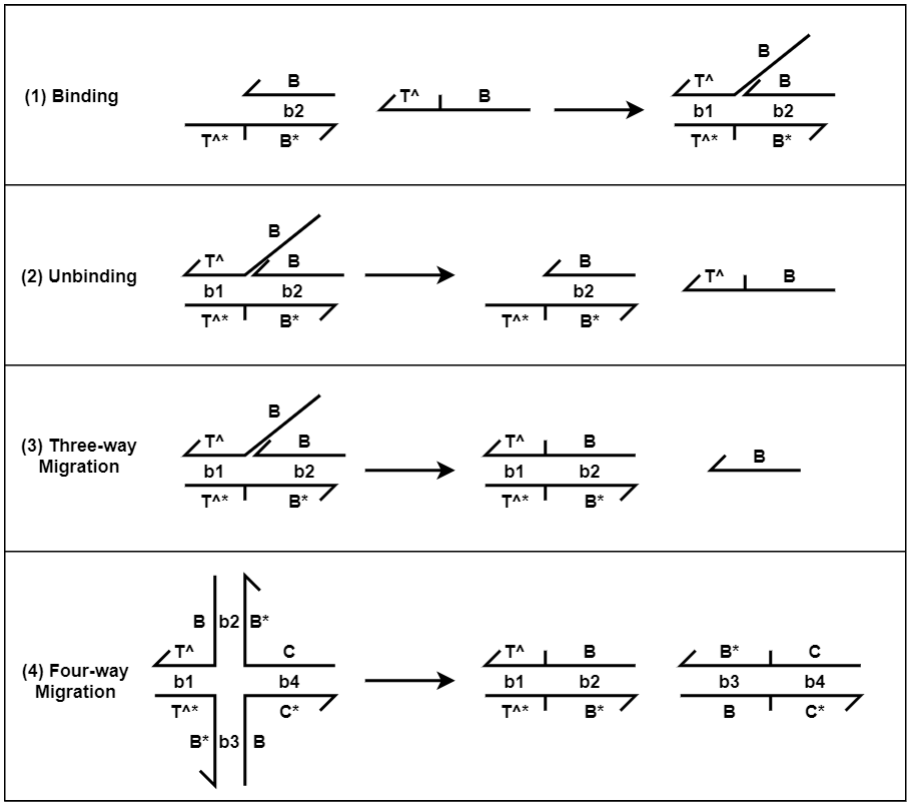



Figure 5: Illustrations of DSD reaction types.

assume $\rho(r)$ is given by:

$$\rho(r) = k \prod_{a \in A} [a] \tag{1}$$

where $k$ is a *reaction constant* and $[a]$ denote the concentration of the species $a$. Each reaction type has a distinct $k$, formulas to derive these reaction constants are given in Section C in [16]. Notice that there are other types of reactions relevant to DSD systems presented in [16] which are not covered by this thesis, we further discuss them as future work in Section 6.2. Among the four reaction types that we consider possible, only binding is dependent on concentrations, and thus have a unit of $M^{-1}s^{-1}$. In the experiments with our tool DSDPy, we use $k = 3 \times 10^5\ M^{-1}s^{-1}$ for binding reactions, other reaction constants are consistent with what was reported by [16].

## 2.3 Rule-Based Modelling

Rule-based modelling was proposed as a computational approach for understanding the physical interactions of biomolecules in cellular regulatory systems, which are commonly studied for cellular signalling [15, 27]. Conventionally, a mathematical model for a biomolecular system is reaction-based, i.e., numerous equations are en-

coded for modelling every possible reaction in the system. Rule-based modelling uses formalised rules to express sets of reactions, each set having its own interpretation. A rule describes certain conditions that the reactants need to satisfy in order to initiate a particular type of chemical transformation. Moreover, rules can be defined and visualised by graphs so that computer scientists and mathematicians do not need to acquire additional knowledge to implement a rule-based model [20].

A model, in any case, should provide details of the quantities of each tractable component in a system and reveal the logical consequences of the system to guide experimentation. A direct challenge comes from this exploration of the system: the number of components in a system may increase exponentially. In practice, thousands of chemical species can be generated by even a small number of proteins [10]. Hlavacek et al. [19] referred to this challenge as *combinatorial complexity*. To address this challenge, dynamic models that account for all possible species can help by deciding if a species is favoured to be traced. It was discovered that usually only a small amount of species are efficiently populated in these models and the population counts can change dramatically if the system dynamics (e.g., concentrations) is changed [12]. Therefore, one can tune the system dynamics so that the favoured species are populated as desired and the unfavoured are suppressed. In the modelling process, only reactions with populated reactants are considered feasible, and thus an explosion of generated species may be prevented.

To specify a model, one needs to provide the knowledge and assumptions about a system so that a mathematical analysis can be pursued. Traditionally, one needs a so-called *reaction-scheme diagram* to specify the possible species and reactions in the system and then needs to translate this diagram into a system of coupled ordinary differential equations (ODEs) [47]. Rule-based models take various approaches for model specification. We introduce the approach adopted by BioNetGen [18] in the remaining part of this section, as we use some of its concepts and implementations for RuleDSD.

**BioNetGen (BNG)** BioNetGen is a software for generating and simulating rule-based models [18]. It is used for modelling various biochemical processes such as gene regulation and metabolism [6, 5] in addition to cellular signalling. The role that the rules in BNG play is described as follows. A rule is written in a similar format as a chemical reaction, specifying a pattern that the reactant(s) and product(s) should match. Given a set of initial species, each rule is examined for its matching reactants. A reaction is established once the pattern matching succeeds, and then the corresponding product(s) are added to the pool of possible species. Each reaction is assigned a reaction rate which is associated with the corresponding rule.

BNG supports mainly two types of simulation [41]: deterministic simulation and stochastic simulation. A deterministic simulator takes in the concentrations of initial species and uses numerical integration of ODEs to model the system through time. This setting is based on the assumption that the system is in an isothermal reactor of

constant volume and the notion of an individual molecule does not exist, which makes the trajectories of changes in concentrations smooth and continuous [41]. However, the deterministic simulator does not work well when the concentrations are small, because then stochastic noise becomes important, i.e., a single molecule can affect the system in a significant way. A stochastic simulator solves this problem by performing the simulation using Gillespie's stochastic simulation algorithm (SSA) [14], which updates one reaction at a time. We do not elaborate on the implementation details of simulators in this thesis, because RuleDSD does not implement any simulator of its own, instead it uses simulation engines available through the PySB [26] framework, a Python package that supports initialization and simulation of customisable rule-based models.

Two approaches can be used to define the relation between the generation and the simulation of reaction networks: "generate-first" approach and "on-the-fly" approach [20]. As the names suggest, the generate-first approach generates all possible species and reactions in advance of a simulation; whereas the on-the-fly approach generates and simulates at the same time, which may be useful in addressing the problem of combinatorial complexity when the reaction network is large or unbounded [20]. With the on-the-fly approach, reactions are generated only if their reactants become populated, which means that some species may not be tracked because they are not produced. BNG originally used the generate-first approach, then updated to the on-the-fly approach [11]. RuleDSD adopts the generate-first approach, as it mainly aims at revealing the interactions between the molecules in a DSD system. Hence, RuleDSD generates reaction networks following the general rule-based modelling concept, and then simulates the networks using existing simulation engines.

# 3 Design

In this section, we briefly introduce the structural design of the RuleDSD pipeline, which contains four major components. We further discuss three critical concepts used in RuleDSD regarding the graphical representations of domain-level DNA strand complexes, which we term *species*.

## 3.1 Overview

RuleDSD is a pipeline that generates and simulates complete reaction networks of DSD systems. The central idea of RuleDSD is to enumerate all species reachable from a given set of species by applying the DSD rules and to encapsulate the discovered species and explored reactions in a PySB model for further analyses. By adopting the rule-based modelling approach, RuleDSD uses graphical structures to represent species and defines rules as the types of reactions in DSD systems. These rules are logical checks that can be performed on graphs, and if the checks of a rule are passed for some reactant species, we say that the rule can be applied and a reaction of this particular type is feasible. Algorithm 1 presents the implementation of the enumerator underlying RuleDSD.

RuleDSD provides two types of integrators for DSD system simulation based on the PySB model, SciPy ODE integrator and BioNetGen integrator. RuleDSD produces not only lineplots of time-course data from simulations, but also species netlists and reactions in a textual format that can be exported and saved by users. In accordance with the functions that RuleDSD provides, the software design pipeline consists of four components as illustrated in Figure 6.

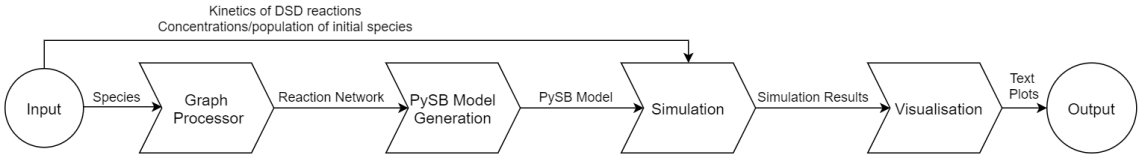



Figure 6: The RuleDSD pipeline design.

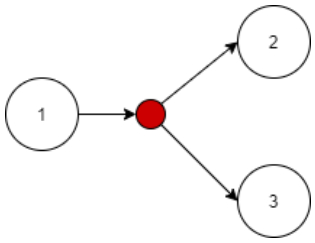
**Graph Processor** The graph processor plays the most critical role in the pipeline as it generates complete DSD reaction networks. The enumeration of all the species reachable from the initial species in a DSD system can be viewed as a loop that terminates when there are no new species to be discovered, and each iteration in the loop is a step attempting to find new species by updating the reaction network.

One iteration of the algorithm that the graph processor executes can be roughly divided into two phases, *Generation* and *Mapping*. The generation phase expands the current reaction network by applying the DSD rules, whereas the mapping phase performs a check on the newly discovered species in the generation phase to see if they

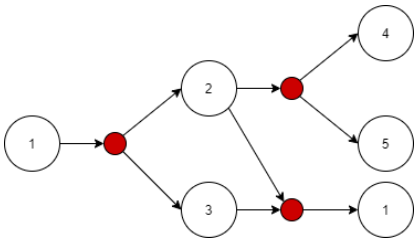
have already been included in the current reaction network. The current reaction network is only updated when both generation phase and mapping phase have finished.

Figure 7 shows an example of an iterative process underlying graph processor in which new species are generated and mapped to the existing species so as to produce a reaction network. The generation phase explores all possible intra-species and inter-species reactions given the species at hand. In our example, species 1, 2, 3 in Figure 7 (a) are the reactant species that we want to put into the graph processor for a reaction network generation. Once a reaction is accepted as a possible reaction, the current reaction network is expanded by adding the reaction. Note that the graph processor does not include the newly discovered species (e.g., species 4, 5) in the current generation process. The generation process finishes when there is no possible reaction to be added.

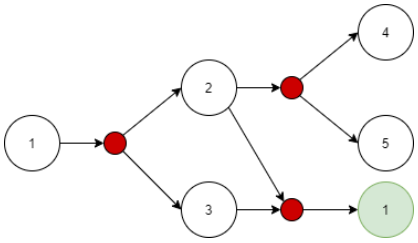
In practice, the mapping algorithm works within the generation process. Specifically, once a reaction is added to the reaction network, the mapping algorithm checks if the product(s) of the reaction have been discovered before. Species 4 and species 5 in Figure 7 (c) are indeed new species in the reaction network, thus, they are stored as newly discovered species. However, species 1 in Figure 7 (c) as the product of the reactants species 2 and species 3 is one of the species that already exists. Therefore, the product is linked back to the known species 1 instead of storing its another copy as illustrated in Figure 7 (d). The update of a reaction network finishes when both the generation process and the corresponding mapping check of the last added



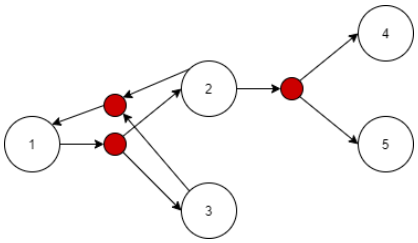

(a) The current reaction network.




(b) The reaction network after the generation phase.




(c) The reaction network in the mapping phase.




(d) The updated reaction network.

Figure 7: An example of species generation and mapping in the graph processor. The vertices denote species and the arrowed edges with a red circle denote reactions. Note that different species are indicated by different numbers.

reaction finishes. In our example, this iteration is finished with species 4 and species 5 as new species.

The next iteration goes on with the updated reaction network in Figure 7 (d) as the current network and the graph processor explore now species 1, 2, 3, 4, 5. This loop ends when one iteration with no new species is finished. A pseudocode code of this graph processing method is shown in Algorithm 1.

**Alorithm 1** Generate New Species

**Input:** $InitSL$: List of Initial species
**Output:** $SL$: List of all possible species

**function** GENERATION($InitSL$)
  $SL = InitSL$
  $visited[i]$ = **False for** $i = 0$ **to** $len(SL) - 1$
  $cursor = 0$
  **while not** $visited[cursor]$ **do**
    $oldlen = len(SL)$
    **for** $i = cursor$ **to** $oldlen - 1$ **do**
      $SL.insert(check\_mono\_reaction(i))$ ▷ $check\_mono\_reaction(i)$ returns a list of all possible species generated by inter-species reaction concerning the species with index $i$ in $SL$.
      $visited[i] =$ **True**
    **end for**
    $comb \leftarrow$ combinations of old species (whose index $i$ in $SL$ satisfies $i < oldlen$) and new species (whose index $i \geq oldlen$)
    **for** $j \in comb$ **do**
      $SL.insert(check\_bi\_reaction(j))$ ▷ $check\_bi\_reaction(j)$ returns a list of all possible species generated by intra-species reaction concerning the species in combination $j$.
    **end for**
    **if** $oldlen \neq len(SL)$ **then**
      $cursor = oldlen$
    **else**
      $cursor = oldlen - 1$
    **end if**
    $newlen = len(SL)$
    **for** $i = oldlen$ **to** $newlen - 1$ **do**
      $visited.insert($**False**$)$
    **end for**
  **end while**
  **return** $SL$
**end function**

**PySB Model Generation** There are three main types of components in PySB models: *Monomer*, *Rule* and *Parameter*. Monomers are the elements whose behaviours one wants to model; rules define chemical reactions between monomers and their complexes; parameters are constant numerical values that represent chemical kinetics rates of the reactions. In RuleDSD, we consider species as an equivalent to the PySB monomers, DSD reactions between the species as the PySB rules, and kinetic rate constants of the DSD reactions as the PySB parameters. Hence, with the information of a complete reaction network provided by the graph processor, RuleDSD can simply construct a PySB model.

**Simulation** PySB provides an interface for numeric integration of PySB models. The supported integrators include the ODE solvers listed in SciPy, the BioNetGen integrator and the Kappa integrator. RuleDSD uses the BioNetGen integrator as primary integrator and the SciPy `lsoda` integrator as a secondary integrator.

**Visualisation** RuleDSD produces not only simulation results as lineplots, but also complete lists of discovered species and explored reactions by the graph processor. It uses the MatPlotLib library as the graph engine for plotting the data retrieved from simulation engines.

## 3.2 Preliminaries

Here we give definitions of the important concepts related to the representations of DNA strand complexes in the RuleDSD modelling approach. We use some common notations from graph theory.

### 3.2.1 Strand Graph

A *Strand Graph* (SG) is a hybrid graph that describes a system of DNA strands. We take the definition of strand graph from [30] with an added attribute *domain*.

**Definition 1.** *A Strand Graph is a 7-tuple* $SG = (V, length, colour, A, toehold, E, domain)$, *where:*

- $V = \{1, ..., N\}$ *is the set of 'vertices', where each vertex is assigned a unique natural number.*
- *'length' is a function that assigns a natural number to a vertex, e.g.,* $length(X) = L$ *indicates that vertex* $X$ *has length* $L$.
- *'colour' is a function that assigns a natural number to a vertex, e.g.,* $colour(X) = C$ *indicates that vertex* $X$ *has colour* $C$.

- *A and E are sets of sets containing two pairs, each pair consists of a vertex and a position, e.g.,* $\{(X_1, Y_1), (X_2, Y_2)\}$ *denotes an element in A or E where* $X_1, X_2$ *are vertices and* $Y_1, Y_2$ $(Y_1 \leq length(X_1), Y_2 \leq length(X_2))$ *are positions in the vertices, respectively. We call the elements in E 'edges' or 'existing edges' and the elements in A 'admissible edges'.*

- *'toehold' is a function that assigns a boolean value to a set containing two pairs, e.g.,* $toehold(\{(X_1, Y_1), (X_2, Y_2)\}) =$ ***True*** *indicates that the set* $\{(X_1, Y_1), (X_2, Y_2)\}$ *is a toehold.*

- *'domain' is a function that assigns an expression (a string of characters) to a pair, e.g.,* $toehold((X_1, Y_1)) =$ *'D' implies that the position* $Y_1$ *in vertex* $X_1$ *is textually expressed as D.*

In RuleDSD design, vertices in a strand graph denote strands. Since RuleDSD aims to model domain-level DSD systems, the smallest units that compose the strands are domains. Therefore, the lengths of vertices are represented in the length of domains. Colours of vertices represent types of strands; strands with exactly the same sequence of domains are considered of the same type of strands. Edges (existing edges) denote bonds between domains, i.e., the edge $\{(X_1, Y_1), (X_2, Y_2)\}$ denotes a bond between domain at position $Y_1$ of strand $X_1$ and domain at position $Y_2$ of strand $X_2$. Namely, a pair of a vertex and a position represents a domain. The domain position is always numbered from the 5' end to the 3' end of the DNA strand, and we start the numbering from 1 in this thesis. Admissible edges denote potential bonds between domains, thus implying that $A \supseteq E$. Toeholds verify the toehold type bonds, i.e., the bond $\{(X_1, Y_1), (X_2, Y_2)\}$ is a toehold type bond if and only if $(X_1, Y_1)$ and $(X_2, Y_2)$ are both toehold domains. Domains denote the expressions of domains in the strands. An example of the strand graph representation is illustrated in Figure 8.

### 3.2.2 Bond Graph

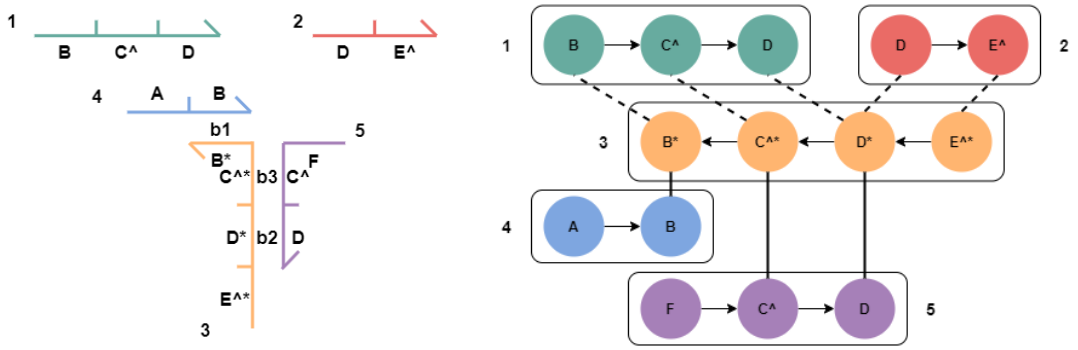
A *Bond Graph* (BG) is a secondary undirected graph derived from an SG that describes the connections between the strands in the SG.

**Definition 2.** *A Bond Graph is a 3-tuple* $BG = (V, colour, E)$*, where:*

- $V = \{1, ..., N\}$ *is the set of 'vertices', where each vertex is assigned a unique natural number. V is inherited from an SG.*

- *'colour' is a function that assigns a natural number to a vertex, e.g.,* $colour(X) =$ *Y denotes vertex X has colour Y. The function colour is inherited from an SG.*

- *E is a set of sets containing two pairs, each pair consists of a vertex and a list of positions, e.g.,* $\{(X_1, [Y_1]), (X_2, [Y_2])\}$ *denotes an element in E where* $X_1, X_2$ *are vertices and* $Y_1, Y_2$ *are positions in the vertices. We call the elements of E 'edges'.*




(a) Schematic of a domain-level DSD system.

(b) Strand Graph of (a).

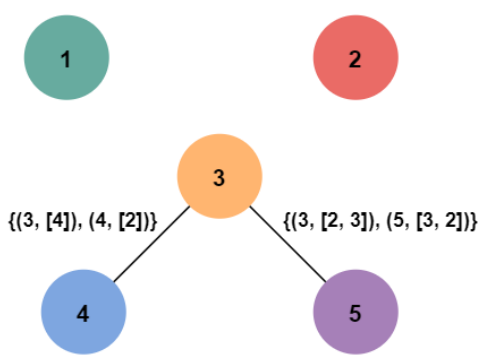
V = {1, 2, 3, 4, 5}
length = {1 → 3, 2 → 2, 3 → 4, 4 → 2, 5 → 3}
colour = {1 → 1, 2 → 2, 3 → 3, 4 → 4, 5 → 5}
A = {{(1, 1), (3, 4)}, {(1, 2), (3, 3)}, {(1, 3), (3, 2)}, {(2, 1), (3, 2)},
{(2, 2), (3, 1)}, {(4, 2), (3, 4)}, {(5, 2), (3, 3)}, {(5, 3), (3, 2)}}
E = {{(4, 2), (3, 4)}, {(5, 2), (3, 3)}, {(5, 3), (3, 2)}}
toehold = {{(1, 2), (3, 3)} → True, {(2, 2), (3, 1)} → True,
{(5, 2), (3, 3)} → True, others → False}
domain = {(1,1) → 'B', (1,2) → 'C^', (1,3) → 'D', (2,1) → 'D', (2,2) → 'E^',
(3,1) → 'E^*', (3,2) → 'D*', (3,3) → 'C^*', (3,4) → 'B*', (4,1) → 'A',
(4,2) → 'B', (5,1) → 'F', (5,2) → 'C^', (5,3) → 'D'}

(c) Textual representation of (b).

Figure 8: An example of a DSD system represented in strand graph notations. The '$*$' symbol denotes complementary domains and the '^' symbol denotes toehold domains. The colour numbers 1, 2, 3, 4, 5 denote respectively green, red, yellow, blue, purple in (a) and (b). We use solid lines and dotted lines to represent edges and admissible edges, respectively. Note that the arrows in (b) only demonstrate the orientations of domains on strands from the 5' end to the 3' end.

The idea of a bond graph is to simplify an SG while preserving its central parts for calculations. RuleDSD works with domain-level binding and unbinding of strands in DSD systems. Hence, it only needs information on the connectivities of domains between strands in the system.

A BG inherits the basic elements $V$ and *colour* from an SG, they represent strands and strand types as they do in the corresponding SG. Since only the connectivities are relevant, the positions of bonded domains are needed instead of the details of domains, because they show how the strands are connected. In contrast with an SG, an element of an edge in a BG does not represent one bond, but all the bonds between the two bonded strands. As shown in Figure 9 (a), the edges in BGs denote the connections between strands. For example, the edge $\{(3, [2, 3]), (5, [3, 2])\}$ denotes that strand 3 is bonded to strand 5 by the domains at position 2 and position 3 of strand 3, whose corresponding bonded domains are at position 3 and position 2 of strand 5, respectively.




(a) Bond graph.

V = {1, 2, 3, 4, 5}
colour = {1 → 1, 2 → 2, 3 → 3, 4 → 4, 5 → 5}
E = {{(4, [2]), (3, [4])}, {(5, [2, 3]), (3, [3, 2])}}

(b) Textual representation of (a).

Figure 9: Bond Graph derived from the Strand Graph in Figure 8.

Note that RuleDSD does not check if the positions of domains in the edges of a BG are valid, because the BG does not have the attribute *length*. This may lead to inaccuracies when it relies on BGs for further calculations. Therefore, it is strictly defined in RuleDSD that a BG must have a parent SG which it represents, and such a relation is one-to-one in the system.

### 3.2.3 Canonical Labelling of Species

*Species* are DNA molecules consisting of interconnected strands. As a special case, we also consider a strand with no connections with other strands a species. For example, in Figure 9, the combination of strands 3, 4, 5 is a species, whereas strand 1 and strand 2 are two other species.

RuleDSD usually uses a strand graph to represent a species because it contains full information. However, a strand graph may contain several species as the edges may not connect all the vertices. Thus, a partition of a strand graph is needed to divide components of strand graphs, each represents only one species. We present this algorithm in Section 4.2 and we assume that every strand graph discussed in this section represents only one species.

In this thesis, we represent species textually by adopting the syntax of *process calculus* from [30]. We apply slightly different naming as given by [30] in the following definition.

**Definition 3.** *Syntax for textual representation of species, in terms of domain name*

*$x$ and bond name $i$.*

| | |
|---|---|
| *Domain* $d ::= x$ | *Domain name* |
| $\mid x^*$ | *Complementary domain* |
| $\mid \hat{x}$ | *Toehold domain* |
| *Possible bonded domain* $o ::= d$ | *Free domain* |
| $\mid d!i$ | *Bonded domain with bond name* $i$ |
| *Strand* $S ::= o_1 ... o_N$ | *Sequence of domains,* $N \geq 1$ |
| *Species* $P ::= \langle S_1 \rangle \mid ... \mid \langle S_N \rangle$ | *Multiset of strands,* $N \geq 1$ |

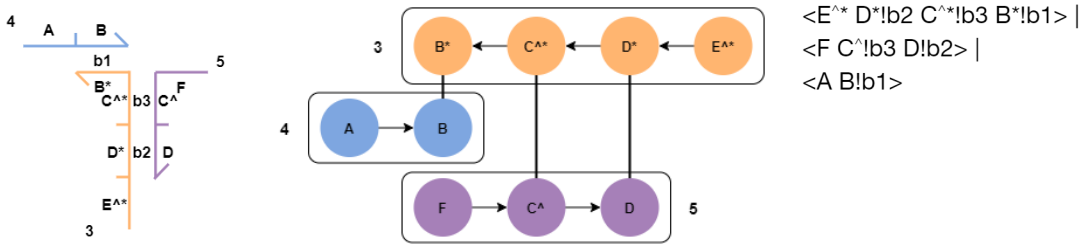
We call this textual representation of a species a *labelling* due to the fact that it can be seen as an assignment of labels to a strand graph (See Figure 10). A labelling of a species given its strand graph can be derived by a graph traversal. Hence, a simple Breadth-First Search (BFS) algorithm is used to derive the labelling of a species as illustrated in Algorithm 3.

We briefly discuss the labelling algorithm (Algorithm 3) by a walkthrough using the example in Figure 10. Suppose we start from the strand 3 in (b), we write the domains occurred in the strand from the 5' end to the 3' end and wrap them with brackets. When there is a bonded domain occurred, we write a '!' symbol to indicate that it is bonded and append its bond name afterwards. Nevertheless, there must be two bonded complementary domains with the same bond name in a labelling, with the restriction that this bond name occurs only twice in the labelling. Therefore, we represent strand 3 in a labelling as $\langle E\hat{}^* \ D^*!b2 \ C\hat{}^*!b3 \ B^*!b1 \rangle$. We then append a '|' symbol as a separation of strands, and write the first strand that strand 3 binds with, i.e., the strand with a domain that binds with the smallest domain in strand 3, which is strand 5 in this case. We move on to the second strand that strand 3 binds with (strand 4), and we finish this exploration of strand 3 when all the strands that bind with strand 3 are written. We further explore the next written strand (strand 5 in our example), and so on. The derivation is finished when all the strands in the species are written.

Recall that there is a mapping phase in the graph processor of RuleDSD in which it detects if a newly discovered species coincides with known species. One can consider this problem as a graph isomorphism problem, where one needs to tell if a given strand graph is isomorphic to any of the existing strand graphs. The definition of a strand graph isomorphism is similar to that of a standard coloured graph isomorphism, with added features concerning the positions on the vertices.

**Definition 4.** *Let $G$ and $G'$ be strand graphs. An 'isomorphism' is a bijective function $\sigma : G.V \rightarrow_{bij} G'.V$ satisfying the condition that $\sigma(G) = G'$, i.e.:*

1. $\forall v_1, v_2 \in G.V, i \in [1, G.length(v_1)], j \in [1, G.length(v_2)]$, *it holds that* $\{(v_1, i), (v_2, j)\} \in G.E \iff \{(\sigma(v_1), i), (\sigma(v_2), j)\} \in G'.E$.




(a) Schematic domain-level representation.

(b) Strand graph representation.

(c) Labelling representation.

Figure 10: Representations of an example species taken from Figure 8. The labelling in (c) is derived by Algorithm 3 given (b) and start vertex 3 as input.

2. $\forall v \in G.V$, *it holds that* $G.colour(v) = G'.colour(\sigma(v))$.

*An isomorphism from $G$ to itself is called an 'automorphism'.*

Since strand graph isomorphism is an edge-preserving bijection with the additional information on the edges including the coloured vertices and positions in vertices, two strand graphs are isomorphic to each other if and only if they represent the same species.

In graph theory, the canonisation problem for finding a canonical labelling of a graph is often connected to the graph isomorphism problem. A canonical labelling of graph $G$ is an isomorphism-invariant labelling of $G$'s vertices, i.e., the canonical labellings of two graphs are the same if and only if they are isomorphic to each other.

**Definition 5.** *A canonical labelling $L(G)$ of a graph $G$ satisfies the property that for any graph $G'$, $L(G) = L(G') \iff G \simeq G'$.*

Algorithms for canonical labelling of graphs used for modelling in biology have been well studied [29, 23]. Inspired by the canonical labelling algorithms given in [29], we prove the existence of a canonical labelling for a strand graph that represents a species in the rest of this section. With the canonical labelling of strand graphs, RuleDSD creates a tag for a strand graph that allows it to compare the tag with other tags so that it learns if the strand graph represents the same species with other strand graphs, which is exactly what should be achieved in the mapping phase.

**Theorem 1.** *Given two strand graphs $G, G'$ with starting vertices $s, s'$ to Algorithm 3, the outputs are the same if and only if there exists a strand graph isomorphism $\sigma$: $G \simeq G'$ such that $\sigma(s) = s'$.*

*Proof.* We first show that Algorithm 3 producing the same outputs for $G$ and $G'$ implies that $G$ and $G'$ are isomorphic to each other. To further analyse the statement, we give Algorithm 2, a method to rename the vertices in $G$ and $G'$ and to produce a

**Alorithm 2** Define $\sigma$

**Input:** $ls$: a labelling derived from G; $ls'$: a labelling derived from G'.
**Output:** $\sigma$: a bijective function.

1: **function** DERIVATION($ls, ls', G, G'$)
2: $\alpha, \alpha' = \emptyset$
3: $ls.split('|')$ ▷ Split the string $ls$ into lists using character '|' as separators
4: $ls'.split('|')$
5: **for** $i = 1$ **to** $ls.length$ **do**
6: $\alpha = \alpha + \{i\}$
7: $\alpha' = \alpha' + \{ls.length + i\}$
8: $\sigma(i) = ls.length + i$
9: $\sigma(ls.length + i) = i$
10: **end for**
11: $G.V = Rename(G.V, \alpha)$ ▷ Rename elements in G.V according to $\alpha$
12: $G.V' = Rename(G.V', \alpha')$
13: **return** $\sigma$
14: **end function**

function serving as a strand graph isomorphism.

Algorithm 2 renames the vertex in position $i$ of $ls$ as $i$ and the vertex in position $i$ of $ls'$ as $ls.length + i$ and sets the bijection $\sigma$ between $i$ and $ls.length + i$ for $i \in [1, ls.length]$. Since the outputs are the same, the sets of colours of vertices and the sequences these vertices appear in the labellings are the same. Therefore, we have $\forall v \in G.V,\ G.colour(v) = G'.colour(\sigma(v))$.

The canonical form includes the binding information between vertices, that is, $\forall p \in [1, ls.length]$, the vertex $v_p$ at position $p$ in $ls$ and the vertex $v_{ls.length+p}$ at position $p$ in $ls'$ are bonded to the vertices at the same positions $k_1, .., k_n$ $(n \geq 1)$ in $ls$ and $ls'$ at the same positions of domains, respectively. Therefore, we have $\forall t \in [1, n], i \in [1, G.length(v_p)], j \in [1, G.length(v_{k_t})]$:

$$\begin{aligned} \{(v_p, i), (v_{k_t}, j)\} \in G.E &\iff \{(v_{ls.length+p}, i), (v_{ls.length+k_t}, j)\} \in G'.E \\ &\iff \{(\sigma(v_p), i), (\sigma(v_{k_t}), j)\} \in G'.E. \end{aligned}$$

We conclude that $\sigma$ is an isomorphism, and we have $G \simeq G'$. We check that it is indeed the case that $\sigma(s) = s'$. Therefore, if Algorithm 3 produces the same outputs for $G, G'$, then $G, G'$ are isomorphic to each other.

Now we show that the existence of an isomorphism $\sigma : G \simeq G'$ implies that Algorithm 3 produces the same outputs for $G, G'$ given the starting vertices $s, s'$ where $\sigma(s) = s'$. Assume that we give $s, G$ and $s', G'$ separately to Algorithm 3, and we use a prime symbol to denote values obtained in the run with $s', G'$, e.g., $ls'$ denotes the output of the run with $s', G'$. We can see from the inner loop (lines 13 to 29) of Algorithm 3 that for each vertex $v$ ($v'$ if in the run with $s', G'$) added to queue Q, it is the

case that $\sigma(v) = v'$. Therefore, each vertex in the sequence $ls$ has the same string representation of that in the sequence $ls'$, implying that $ls = ls'$. □

**Corollary 1.** *There exists a canonical labelling, if one can establish that for any two isomorphic strand graphs $G \simeq G'$, it is the case that $\sigma(s) = s'$ for the starting vertices $s, s'$ of Algorithm 3.*

*Proof.* We know from Theorem 1 that two labellings derived by Algorithm 3 are the same if and only if it holds that $\sigma : G \simeq G'$ such that $\sigma(s) = s'$. Therefore, a labelling is unique for an isomorphism class if the starting vertices chosen for this class are isomorphic to each other. □

We further discuss the derivation of a canonical labelling in Section 4.2 together with optimisation techniques available to speed up the canonical labelling process.

**Alorithm 3** Derive a Labelling

**Input:** $s$: start vertex; $SG$: strand graph of a species.
**Output:** $ls$: a labelling of the species.

```
1: function DERIVE_LABELLING(s, SG)
2:     for u ∈ SG.V − {s} do
3:         u.visited = False
4:     end for
5:     s.visited = True
6:     ls = String()
7:     Q = ∅
8:     ENQUEUE(Q, s)
9:
10:    while Q ≠ ∅ do
11:        u = DEQUEUE(Q)
12:        str =′ ⟨′
13:        for i = 1 to SG.length(u) do
14:            str = str + SG.domain((u, i))
15:            if is_bonded(u, i) then
16:                (v, j) = bonded_to(u, i)
17:                if v.visited = False then
18:                    str = str+′!′ + create_bond_name(u, i, v, j)        ▷
   create_bond_name() simply name the bond by the order that it is encountered.
19:                    ENQUEUE(Q, v)
20:                else
21:                    str = str+′!′ + get_bond_name(u, i)
22:                end if
23:            end if
24:            if i = SG.length(u) then
25:                str = str+′⟩′
26:            else
27:                str = str +′ ′
28:            end if
29:        end for
30:        u.visited ← True
31:        if Q.is_empty() then
32:            ls = ls + str
33:        else
34:            ls = ls + str +′ |′
35:        end if
36:    end while
37:    return ls
38: end function
```

# 4 Methods

In this section, we present the technical details of the RuleDSD pipeline. The section is organised as shown in Figure 6, except that we discuss the generation and mapping phase in the graph processor separately in Sections 4.1 and 4.2.

## 4.1 Generation of Reaction Network

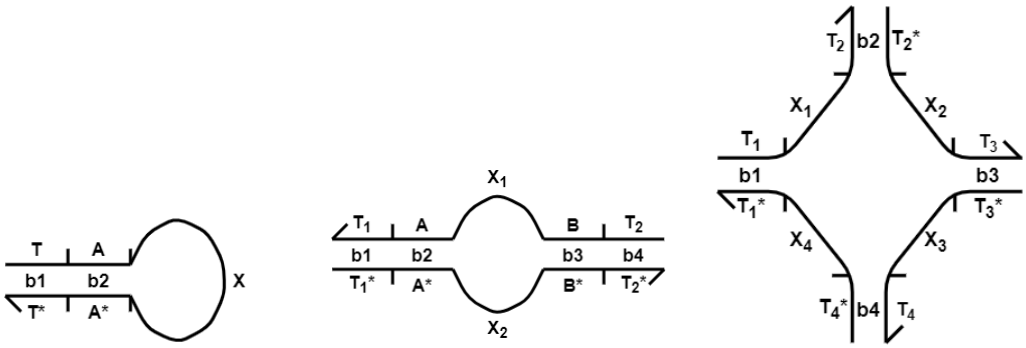
The reactions in a DSD system follow four rules: Rule Binding (RB), Rule Unbinding (RU), Rule Three-Way Branch Migration (R3) and Rule Four-Way Branch Migration (R4). We only consider reactions that conform to one of these rules. Reactions are further categorised into two types: intra-species and inter-species reactions. All four rules are applicable to intra-species reactions, whereas only RB is applicable to inter-species reactions.

As shown in Algorithm 1, the RuleDSD graph processor (referred to as the graph processor in the following sections) enumerates all possible reactions based on the current species, meaning that it checks if the conditions for applying any of the four rules are satisfied for each species and then checks if the combination of any two current species satisfies the conditions of RB. The generation phase starts with a set of initial species provided as user input in a labelling form and finishes when there are no more reactions to be explored. Note that the checks for rules are performed on strand graphs of species or combinations of species. Therefore, we refer 'edges' to edges (existing edges) in strand graphs unless we define the scope of the discussion is with respect to bond graphs in this section. In the case of an inter-species binding check, the two separate strand graphs of the reactants are merged to one strand graph preserving all the details.

When a rule is applied, a reaction is formed and the product(s) of reactant(s) will then be put into the mapping phase. At the same time, the information of the applied reaction is stored so that the full list of reactions can be further retrieved in a PySB model. We discuss the implementations of the four rules in the following subsections and the mapping algorithm in Section 4.2.1.

### 4.1.1 Rule Binding (RB)

Given a strand graph, the graph processor examines each admissible edge that is not an existing edge ($\forall e \in A - E$) eligible for binding. To check if two domains in an admissible edge can be bonded together, the graph processor first needs to decide if they belong to the same species. If the domains belong to the same species, they must satisfy the following preconditions for binding: 1) both domains are *free*, i.e., they are not bonded to any other domains; 2) the two domains are not *hidden* in different loops of bonds (see Figure 11 for examples); 3) the binding is antiparallel.




(a) Hidden domain in a harpin loop.

(b) Hidden domains in two strands.

(c) Hidden domains in multiple strands.

Figure 11: Illustration of hidden domains. We say domain $X$ in (a), domains $X_1, X_2$ in (b) and domains $X_1, X_2, X_3, X_4$ in (c) are *hidden* and thus they are not available for binding with other domains outside the loops.

If the domains do not belong to the same species, they should satisfy the additional precondition such that they are toehold domains.

The condition difference between intra-species and inter-species binding is because the execution of a DSD system is designed around a set of toehold-mediated strand displacement steps, where inter-species binding reactions mainly involve short toehold domains. Inter-species binding reactions between long domains are traditionally discarded in the design phase, as they do not serve any useful purpose in the DSD systems.

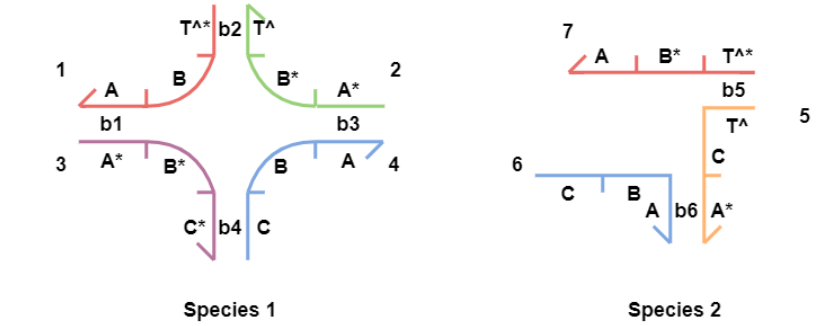
**Intra-species Binding & Inter-Species Binding** RB is involved in both intra-species and inter-species reactions, and the two types of reactions shall be treated separately. Here, we discuss how to categorise a binding as intra-species or inter-species in a strand graph that contains more than one species.

Finding the number of species in a strand graph is equivalent to finding the number of connected components in the corresponding bond graph. Therefore, the problem can be viewed as finding a *spanning forest* of a bond graph. We give the definitions of spanning tree and spanning forest as follows.

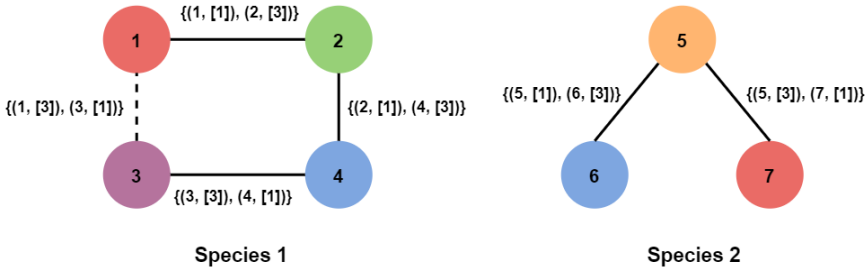
**Definition 6.** *A graph $F = (V', E')$ is a spanning forest of a graph $G = (V, E)$ if $V' = V$, $E' \subset E$, and each component of $F$ is a spanning tree of a component of $G$ [2].*

**Definition 7.** *A graph $T$ is a spanning tree of an undirected graph $G$ if $T$ is a tree which includes all of the vertices of $G$.*

One can use any graph traversal algorithm iteratively to obtain a spanning forest of a given bond graph. Hence, one can store the covered vertices in one iteration




(a) Schematic domain-level representation.




(b) Bond Graph representation.

Figure 12: Representations of two example species. In (b), Solid lines represent edges in a spanning forest of the bond graph and dashed lines represent edges that are not in the spanning forest.

as vertices of a species. In the example in Figure 12, suppose we start the traversal from vertex 1 and name the species by the iteration count, then vertices $\{1, 2, 3, 4\}$ belong to species 1 and vertices $\{5, 6, 7\}$ belong to species 2. Such process can be done when the bond graph is initialised.

When an admissible edge in the strand graph is examined, the graph processor can then see if the vertices on the admissible edge are of the same species by looking into the stored information in the corresponding bond graph. For example, the admissible edge $\{(1, 2), (3, 2)\}$ in Figure 12 will be categorised as an intra-species binding because vertices $1, 3$ are both in species 1, whereas the admissible edge $\{(1, 2), (6, 2)\}$ will be categorised as an inter-species binding because vertex 1 is in species 1 and vertex 3 is in species 2.

**Hidden Domains** Hidden domains are the domains inside a loop of bonds (see Figure 11). It is not possible for the hidden domains in a loop of bonds to bind with domains that are outside of this loop because the loop forms a rather stable structure. There are many algorithms available for finding loops in a graph, here we give Algorithm 4 based on the spanning forest obtained earlier for loop detection. Therefore, this task is performed on bond graphs. Note that we use adjacency list ($G.Adj$ in Algorithm 4) to represent the edges in the bond graph.

**Alorithm 4** Detect Loop

**Input:** $G$: a bond graph; $T$: a spanning tree of $G$;
**Output:** *loop*: a list of all the loops in $G$.

```
1: function DETECT_LOOP(G, T)
2:     loop = List()
3:     for u ∈ T.V do
4:         for e ∈ G.Adj[u] do
5:             if e ∉ T.Adj[u] then
6:                 v = e.node2                ▷ get the vertex that u is connected to by e
7:                 path = search_path(u, v, G)   ▷ search_path(u, v, G) returns all the paths in lists from u to v in G
8:                 loop.insert(path)
9:             end if
10:        end for
11:    end for
12:    return loop
13: end function
```

**Alorithm 5** Get Hidden Domains

**Input:** $G$: a bond graph; *loop*: a list of all the loops in $G$;
**Output:** *hidden*: a list of all the hidden domain in *loop*.

```
1: function GET_HIDDEN(G, loop)
2:     hidden = List()
3:     for i ∈ loop do
4:         if i == 0 then
5:             (pre, suc) = (len(loop) − 1, i + 1)
6:         else
7:             if i == len(loop) − 1 then
8:                 (pre, suc) = (i − 1, 0)
9:             else
10:                (pre, suc) = (i − 1, i + 1)
11:            end if
12:        end if
13:        e1 = get_edge(i, pre)    ▷ get the edge that connects i to its predecessor
14:        e2 = get_edge(i, suc)    ▷ get the edge that connects i to its successor
15:        d = sort(e1.dom + e2.dom) ▷ sort the bonded domains on i that connects i to its predecessor and successor
16:        for d[0] < j < d[len(d) − 1] do
17:            hidden.insert((i, j))
18:        end for
19:    end for
20:    return hidden
21: end function
```

In short, for each edge in a bond graph that is not an edge of the spanning forest of the bond graph, a pathfinding algorithm is applied to the bond graph (excluding the edge) from one vertex to another in the edge, and the number of paths it returns is the number of loops the edge is involved in. For example, the edge $\{(1, [3]), (3, [1])\}$ in Figure 12 (b) is not an edge of the spanning forest, which indicates that at least one loop exists in the bond graph. Suppose we apply a pathfinding algorithm from vertex 1 to 3, path $[1, 2, 4, 3]$ will be found. This path is then stored as a loop in the bond graph.

The next step is to find and store hidden domains in each loop. Since a loop is stored in an ordered way, the graph processor simply needs to store the domains that are in between the bonded domains that connect a vertex to its neighbours in the loop (see Algorithm 5).

However, the above technique does not deal with hairpin loops and loops between two strands as illustrated in Figure 11 (a) and (b). These two cases are treated separately in RuleDSD. The graph processor can store the hidden domains in these two special cases when the edges of the bond graph are initialised. Once there is an edge that leads to a self-loop, it matches the pattern of a hairpin loop. When there are multiple positions of a vertex connecting to the same vertex, loops between two strands might occur.

**Free Domains** A domain cannot bind with two domains at the same time. Therefore, the graph processor must know if both domains in an admissible edge are free for binding. Such check can be easily done by looking into the edges of the strand graph, i.e., any domain that is contained in an edge is not free.

**Antiparallel** For two strands of nucleotides binding together, the orientations of the sugar-phosphate backbones of the strands have to be opposite to each other. This antiparallel orientation is an important characteristic of a DNA helix, and it allows the nucleotides to complement one another, which makes the helix structurally stable. Since RuleDSD uses domains as the smallest units, it considers antiparallelism at domain-level, i.e., two domains are antiparallel if bonds formed between them satisfy the antiparallel property of the two corresponding strands.

As stated in Section 3.2.1, the positions of a vertex in a strand graph indicate the domain positions in the strand from the 5' end to the 3' end. This universal orientation setting enables the graph processor to check if a binding is antiparallel. In inter-species binding, this check can be skipped because the strands that contain the binding domains are not connected. This allows one of the strands to be placed in either orientation and there is no restriction to place the other strand in the opposite orientation.

We define the notion of *connection path* as follows. It plays a critical role in antiparallelism checking and can be obtained by performing a reformed BFS algorithm on

the bond graph.

**Definition 8.** *A connection path $P$ from a domain $(X, Y)$ to another domain $(X', Y')$ in a strand graph $G$ is a sequence of domains $((X, Y_0), (X_1, Y_1), ..., (X_N, Y_N))$ such that $X_0 = X$, $X_N = X'$, and the set of every two consecutive elements in the sequence correspond to the existing edges of $G$.*

In intra-species binding, the two free domains can not bind together due to a violation of antiparallelism if: 1) for both to-be-bonded domains, each has an adjacent domain and these adjacent domains are in a connection path of the to-be-bonded domains; and 2) the to-be-bonded domains are in the same direction from these adjacent domains. Since the direction can be inferred from the domain position numbers, the graph processor can easily perform this task.

In Figure 13 (a), the admissible edge $e_1 = \{(1, 2), (2, 2)\}$ is not eligible for binding because:

1. Domains $(1, 1)$ (adjacent to domain $(1, 2)$) and $(2, 1)$ (adjacent to domain $(2, 2)$) are in the connection path $((1, 1), (2, 1))$ from domain $(1, 2)$ to domain $(2, 2)$.
2. Domain $(1, 2)$ is in the 3' end direction from domain $(1, 1)$, so is domain $(2, 1)$ from domain $(2, 2)$.

In the contrary to $e_1$, the admissible edge $e_2 = \{(1, 2), (2, 1)\}$ with the connection path $((1, 1), (2, 2))$ from $(1, 2)$ to $(2, 1)$ in Figure 13 (b) is eligible for binding, because

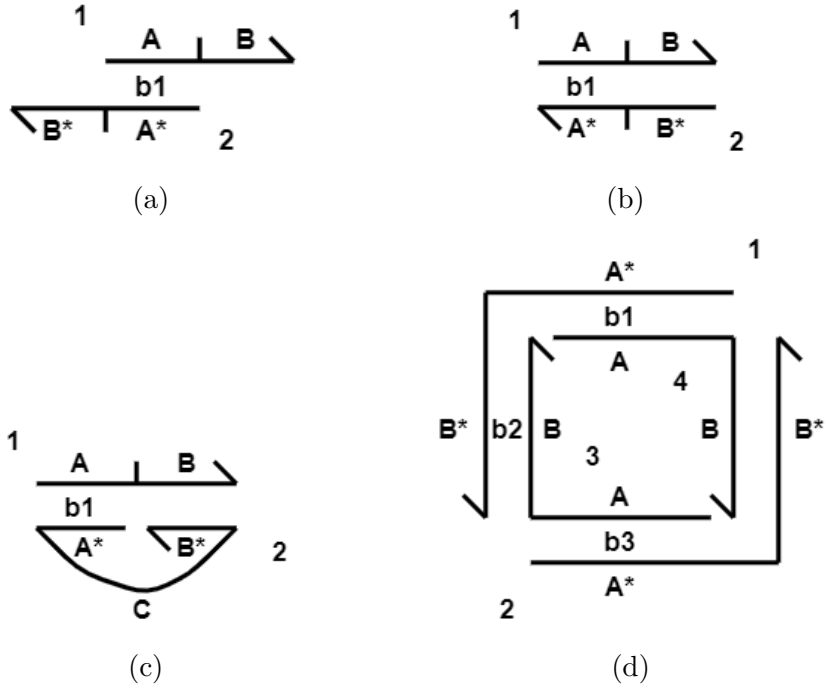



Figure 13: Illustration of antiparallel bindings.

domain $(1, 2)$ is in the 3' end direction from domain $(1, 1)$, whereas domain $(2, 1)$ is in the 5' end direction from domain $(2, 2)$.

Although domain $(1, 2)$ and domain $(2, 2)$ in Figure 13 (a) can not bind together, these two domains can be bonded together provided that free intermediate domain(s) exist in at least one of strand 1 and strand 2 so that a loop can be formed (see (c)). Such looping results in pseudoknotted DNA structures, a class of DNA structures in which non-nested base-pairing occurs. As our named-pairing notation used for representing species does not restrict such DNA structures, the RuleDSD modelling covers DNA structures that may arise due to such looping.

Moreover, the free intermediate domain(s) in between a bonded domain and a to-be-bonded domain can be twisted into any orientations so that the to-be-bonded domain is close enough to bind with another free domain. Note that this twist is feasible if the domain(s) in between is long enough. Since RuleDSD does not include the length of the domain in nucleotides, we choose to set the twist by default feasible.

However, the twists might violate the previously defined anti-parallelism criteria as shown in Figure 13 (d). The admissible edge $\{(2, 2), (4, 2)\}$ is eligible for binding even if:

1. Domains $(2, 1)$ (adjacent to domain $(2, 2)$) and $(4, 1)$ (adjacent to domain $(4, 2)$) are in the connection path $((2, 1), (3, 1), (3, 2), (1, 2), (1, 1), (4, 1))$ from domain $(2, 2)$ to domain $(4, 2)$.

2. Domain $(2, 2)$ is in the 3' end direction from its adjacent domain $(2, 1)$, so is domain $(4, 2)$ from domain $(4, 1)$.

One can easily extend this scenario to other scenarios of a similar system containing even number ($>2$) of strands. The graph processor treats these scenarios as special cases in rule binding. They can be detected by counting the length of the connected path.

**Toehold Binding** Toehold domains are usually short in length of nucleotides such that they can easily bind and unbind with their complementary domains. In the case of inter-species binding, we only consider toehold binding. Recall that there is a toehold attribute in the definition of strand graph in Section 3.2.1, this attribute allows the graph processor to check toehold binding in a convenient way.

### 4.1.2 Rule Unbinding (RU)

The graph processor examines each existing edge ($\forall e \in E$) in a given strand graph eligible for unbinding. If a bond is between two toehold domains (which we call a *toehold type bond*) and its two ends are not held close to each other (e.g., held by an adjacent bond as shown in 14 (a)), the graph processor considers the bond as an

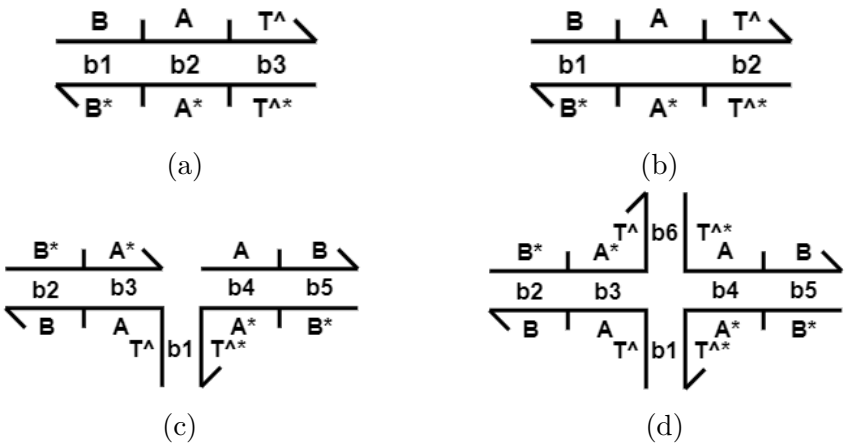



Figure 14: Anchoring of toehold type bonds. (a) Bond $b3$ is anchored because the adjacent domains $A, A^*$ of $T\hat{}, T\hat{}^*$ are also bonded to each other. (b) Bond $b2$ is not anchored because the adjacent domains $A, A^*$ of $T\hat{}, T\hat{}^*$ are free. (c) Bond $b1$ is not anchored because either of its adjacent bonds $b3, b4$ does not attach the undergoing unbinding strands. (d) Bonds $b1, b6$ are not anchored. Although both of their adjacent bonds $b3, b4$ attach the undergoing unbinding strands by a junction $(b3-b1-b4-b6)$, their anchoring here is with respect to toehold type bonds. Namely, there are more than one toehold type bonds in the junction. We consider a toehold type bond as a weaker bond, and thus we say $b1, b6$ are not anchored.

unbinding bond. For example, bonds $b2$ in Figure 14 (b), bond $b1$ in (c) and bond $b1, b6$ in (d) are available for unbinding.

Since the function checking a toehold type bond is already prescribed for rule binding, the graph processor can simply reuse it for rule unbinding.

Peterson et al. [30] describe a bond *anchored* when both ends of the bond are held close to each other. A bond is anchored if one of its adjacent bonds connects the same strands or it is involved in a junction. Notably, if the bond is in a junction, it should satisfy the condition that the junction does not involve any other toehold type bonds (see Figure 14 (d) for a counterexample).

To implement the anchoring inspection feature, the graph processor first checks if there is an adjacent bond holding the two undergoing unbinding strands together and then checks if there is a junction that contains the undergoing unbinding bond when the previous check fails.

The first check for anchoring can be easily viewed from the bond graph. Given unbinding domains $(X_1, Y_1), (X_2, Y_2)$, the graph processor examines the list of domains in vertex $X_1$ that are bonded to domains in vertex $X_2$, $X_1$ is anchored to $X_2$ if $(X_1, Y_1 - 1)$ is bonded to $(X_2, Y_2 + 1)$ or $(X_1, Y_1 + 1)$ is bonded to $(X_2, Y_2 - 1)$.

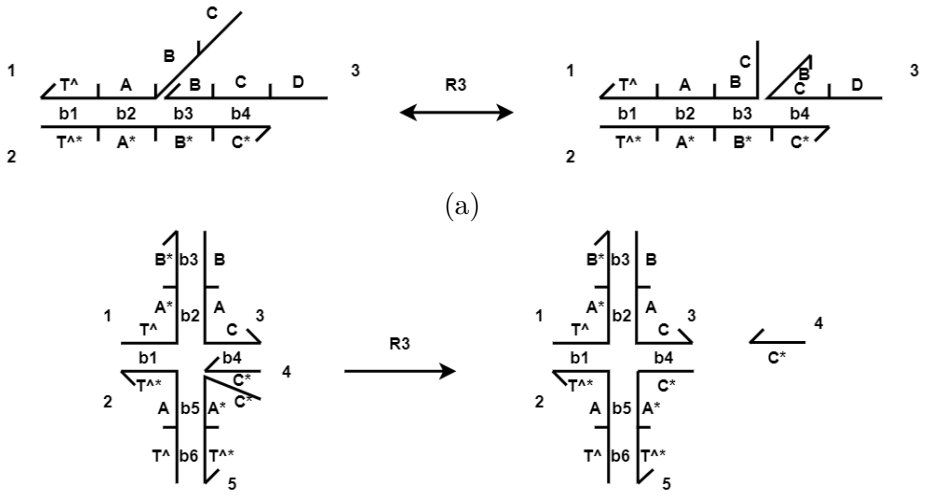
The existence of a junction is the existence of a loop in the bond graph with the condition that the bonds constituting the loop are adjacent to each other. Recall that the graph processor stores the information of all the loops in the bond graph for RB, here it simply reuses that information to test if the undergoing unbinding vertices are in a loop and the edges constituting this loop are adjacent to each other with respect to the positions in each vertex.

### 4.1.3 Rule Three-way Branch Migration (R3)

Three-way branch migration is a process in which an invading strand binds with a base strand and displaces the incumbent strand that is bonded with the base strand, the process is mediated by an already formed adjacent bond (usually toehold type bond) between the invading strand and the base strand. Similar to the rule binding, the graph processor examines each admissible edge that is not an existing edge ($\forall e \in A - E$) in the strand graph eligible for migration (R3 and R4). Particularly for three-way branch migration, one domain in the admissible edge undergoing examination is free and the other is bonded.

Three-way branch migration occurs if: 1) the bond that the admissible edge undergoing examination forms is anchored after the migration occurs, 2) the new bond satisfies the antiparallel property of the DNA complex.

In Figure 15 (a), we see that after the migration occurs, the bond $\{(1,2),(2,3)\}$ is anchored because of $b2$ and satisfies the antiparallel property. Thus, we say that R3 is applicable to the admissible edge $\{(1,2),(2,3)\}$. Similarly in Figure 15 (b), R3 is applicable to the admissible edge $\{(3,3),(5,1)\}$. The bond $\{(3,3),(5,1)\}$ is anchored after migration because the junction involving $b1, b2, b4, b5$ holds, and the strands




(a)

(b)

Figure 15: Illustrations of possible three-way branch migrations.

$3, 5$ are antiparallel.

Notably, after the three-way branch migration occurs in Figure 15 (a), R3 turns applicable to the admissible edge $\{(1, 1), (2, 4)\}$. Considering the adjacency feature of three-way branch migration, the graph processor analyses all the possible following three-way branch migrations together with the first one as a one-step reaction. It means that the product(s) of a three-way branch migration reaction are the final product(s) in the state that no other three-way branch migrations can happen in the direction of the bond that mediated the migration(s).

One can see that the preconditions and postconditions for R3 are functions that we have discussed for RB and RU. Hence, the elaborations for the implementations of R3 are omitted.

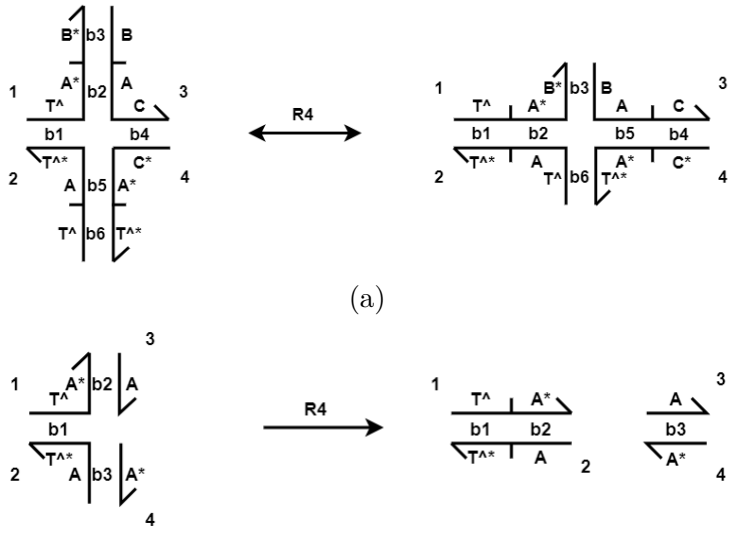
### 4.1.4 Rule Four-way Branch Migration (R4)

Four-way branch migration is a process in which two bonds are simultaneously exchanged between two pairs of complementary domains, mediated by an adjacent bond (usually toehold type) between the exchanging bonds (see Figure 16). The graph processor examines each admissible edge that is not an edge ($\forall e \in A - E$) in the strand graph eligible for R4 as it does for R3. However, both domains on the admissible edge should be bonded in order to initiate four-way branch migration.

Four-way branch migration occurs if: 1) one of the newly formed bonds is anchored after the migration occurs, 2) the two new bonds satisfy the antiparallel property of the DNA complex.

In Figure 16 (a), R4 is applicable to the species on the left-hand side. There are two existing bonds $b2, b5$ that have the same pair of complementary domains and they share the same adjacent bonds $b1, b4$. Therefore, it is feasible for $b2$ and $b5$ to exchange their strands so that there are two new bonds between the two pairs of complementary domains, which are $b2, b5$ in the new species on the right-hand side. Finally, the graph processor checks that the new bonds do not break the antiparallel property in the DNA complex(es), meaning that the new species are feasible. Nevertheless, this particular reaction in (a) is bi-directional because R4 can be applied to the new species, whereas the four-way branch migration in (b) results in two species, and thus is irreversible.

Four-way branch migration can be seen as a special case of three-way branch migration such that there are two invading strands in the former in contrast to one in the latter. The two invading strands are bonded, and at least one of them is able to initiate a three-way migration to form a new bond (the same new bond if it were a four-way branch migration) if it is not bonded. This insight allows the graph processor to use implementations of R3 for R4.




(a)




(b)

Figure 16: Illustrations of possible four-way branch migrations.

## 4.2 Species Mapping

Species are stored as the canonical labellings of their strand graphs in RuleDSD because they enable the graph processor to promptly test graph isomorphism. However, analysing reactions in the DSD system requires strand graph representations, which demands changes between strand graphs and their canonical labellings rapidly.

### 4.2.1 From a Strand Graph to Its Canonical Labelling

As one can tell from the reaction network generation, the reactions rules in a DSD system are based on the strand graphs and we may apply changes to those strand graphs if they satisfy the conditions of a rule. A change in a strand graph leads to a new strand graph which represents one or more new species.

Precisely, the strand graphs that the generation phase produces after applying RU, R3 and R4 may represent more than one species. Since the individual species in the DSD system are the fundamental elements for reactions, it is crucial to separate all the species in these strand graphs. We have shown in Section 4.1.1 how to find the number of connected components (i.e., species) and vertices belonging to each component in a strand graph. In Algorithm 6, we simply retrieve the related vertices of every species in the strand graph and keep every part in the strand graph that represents a species as a new strand graph.

After ensuring that each species has its own strand graph after generation, the graph processor needs a mapping algorithm to determine if the generated species are new

**Alorithm 6** Separate All Species

**Input:** $G$: a strand graph.
**Output:** $lG$: a list of strand graphs.

```
1: function SEPERATE(G)
2:     lG = List()
3:     BG = BondGraph(G)
4:     lsp = BG.get_all_species()      ▷ get_all_species() returns a list of sets of vertices such that each set contains all the vertices in a species.
5:     for i = 1 to lsp.length do
6:         lG = lG + G.keep_vertex(lsp[i])      ▷ keep_vertex(x) returns a strand graph keeping only the vertices in x and their relevant information.
7:     end for
8:     return lG
9: end function
```

**Alorithm 7** Canonical Labeller

**Input:** $G$: the strand graph of a species.
**Output:** $cl$: a canonical labelling of $G$.

```
1: function DERIVE_CANONICAL_LABELLING(G)
2:     l = List()
3:     for v ∈ G.V do
4:         l = l + derive_labelling(v, G)      ▷ derive_labelling() is defined by Algorithm 3.
5:     end for
6:     l.sort()                                ▷ Sort list l with ascending order.
7:     cl = l[0]
8:     return cl
9: end function
```

in the DSD system. By Corollary 1 in Section 3.2.3, we know that there exists a canonical labelling for each species in the DSD system. Namely, two species are the same if and only if their canonical labellings are identical. Hence, the mapping algorithm in RuleDSD simply calls a canonical labeller and compares its output with all the canonical labellings of current species.

We present the canonical labeller for deriving a canonical labelling of a species in Algorithm 7. A species can have at most $N$ different labellings by the labelling derivation algorithm (Algorithm 3) if it consists of $N$ strands, i.e., its strand graph has $N$ vertices. There are many ways to find a unique labelling of a species. Here we suppose that we find the alphabetically smallest labelling. This smallest labelling is a canonical labelling because two labellings are the same if the two strand graphs are isomorphic to each other and two isomorphic strand graphs have the same set of labellings by Theorem 1, meaning that the smallest labelling in the set is identical for the two isomorphic strand graphs.

The canonical labeller given by Algorithm 7 is not optimal as it derives all labellings by using every vertex in the strand graph as a candidate starting vertex. Moreover, there is no need to compare the labellings only when the derivations are finished because the differences may emerge before the traversals end in Algorithm 3. We provide several optimisation techniques for the canonical labeller as follows.

**Candidate Set Cut** The original candidate starting vertex set (*candidate set*) for the canonical labelling includes all the vertices in a strand graph. However, if we always choose a subset of the original candidate set as the candidate set and this exact subset is chosen for every strand graph in an isomorphism class of strand graphs, the alphabetically smallest labelling among the derived labellings using the new candidate set is still a canonical labelling.

For example, we can choose the set of vertices that has the smallest colour as the candidate set. Therefore, the labelling we choose for a strand graph in this setting is the alphabetically smallest labelling that is derived from a vertex of the smallest colour. It is obvious that two isomorphic strand graphs have the same smallest colour, thus, their chosen labellings are the same. We also know that the same two labellings imply graph isomorphism by Theorem 1. Hence, the labelling we choose in this setting is indeed a canonical labelling of the strand graph.

Vigorously, one can choose the set that has the smallest size among all sets of vertices of the same colour as suggested by [30]. In this case, it is possible to skip the comparisons of the labellings in order to obtain the canonical labelling as there may be one unique strand in the species.

One can also impose restrictions other than colour and size on the candidate set. For example, we can prune the set again by the smallest number of edges concerning the vertex. This can be easily achieved by checking the corresponding bond graph.

**On-the-fly Comparison** There are two layers of loops in Algorithm 3, the outer loop (line 10 to 36) sets the current examination position on a specific strand and the inner loop (line 13 to 29) further sets the position on a specific domain on the strand. Even if we choose a set of vertices of the same colour as the candidate set, we might still be able to differentiate the labellings after one iteration of the outer loop. Therefore, we can compare the labellings after each outer iteration finishes in the labelling derivation process, which we call *on-the-fly comparison*. Note that the comparison here is between two strings that may have different lengths, the longer string by convention is larger if the characters of the length of the shorter string in the longer string matches the shorter string. We are choosing the labelling that has the alphabetically smallest string (*sub-labelling*) in every executed outer iteration. This labelling is still a canonical labelling as it is also unique for an isomorphism class.

Suppose we choose $\{1, 2\}$ as the candidate set in Figure 17. After one outer iteration

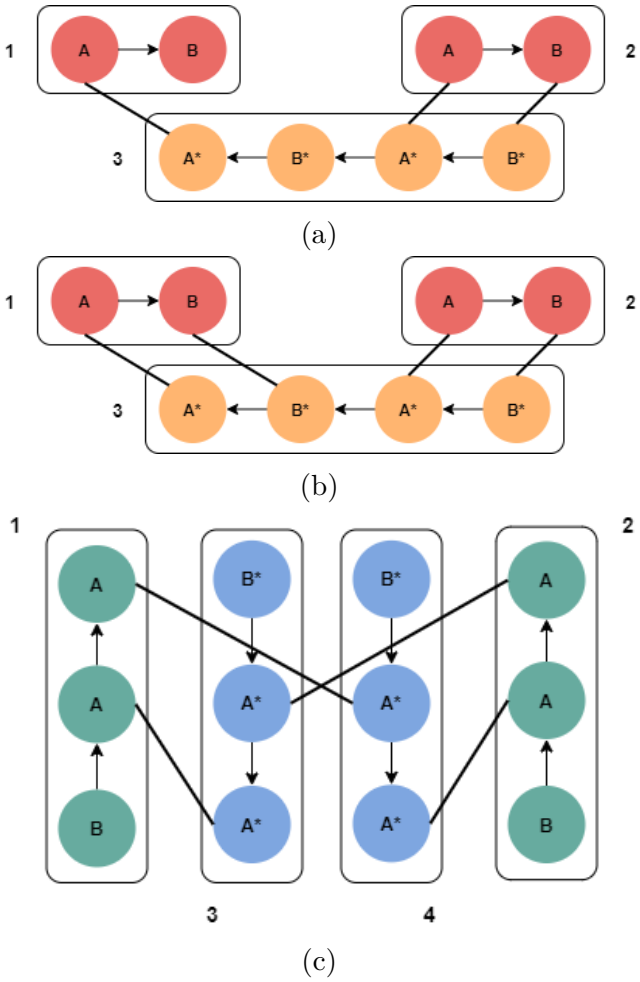



Figure 17: Strand graphs of multiple example species.

in (a), we have sub-labellings $\langle A!1\ B\rangle$ and $\langle A!1\ B!2\rangle$ produced by inputting starting vertices $1, 2$ to Algorithm 3, respectively. The canonical labeller with on-the-fly comparison chooses sub-labelling $\langle A!1\ B!2\rangle$ because the ASCII code of '!' is smaller than '⟩'. Hence, the canonical labelling for the species in (a) is derived using starting vertex 2, which is $\langle A!1\ B!2\rangle|\langle B^*!2\ A^*!1\ B^*\ A^*!3\rangle|\langle A!3\ B\rangle$. In (b), the sub-labellings produced after the first outer iteration are identical. Thus, the second outer iteration is needed to determine which labelling should this species use as its canonical labelling.

However, a situation where there are multiple identical labellings that satisfy the conditions for the canonical labelling may occur (see Figure 17 (c)). This is due to the existence of automorphism in the strand graph and it may occur for all canonical labellers. Nevertheless, we can simply choose either labelling that is left in the end and it is a canonical labelling.

### 4.2.2 From a Labelling to Its Strand Graph

The transformation from a labelling to its strand graph can be seen as the initialisation of the strand graph. That is to say, the strand graph should contain the same information as its corresponding labelling, which is also inferred by the fact that the strand graph and the labelling are two representations used for a species in RuleDSD. In the following, we show that a strand graph indeed contains the information its labelling has.

A labelling $l$ of a species clearly involves the number of strands in the species and the lengths of each strand, which are $V$ and *length* of its strand graph $G$. Since the syntax of labelling $l$ contains the domain-level information including domain name and marks for toehold domains and bonded domains, one can easily obtain functions $domain, toehold$ and the set of edges $E$ for $G$. To construct $A$ for $G$, one needs to match all complementary domains for every domain name in the labelling. A labelling does not explicitly tell the colours of strands it contains. One can keep a hash table globally where the sequence of domains of a strand disregarding the bonds is the index value and the colour it represents is the data value. Therefore, *colour* of $G$ can be retrieved from the hash table by using each strand in the labelling as the index. For simplicity, the graph processor saves the colours of the strands in the labelling together with the labelling itself. Thus, *colour* of $G$ can be retrieved directly.

The above paragraph illustrates the process of deriving a strand graph from a labelling. This process has no ambiguity as there cannot be two strand graphs with any different attribute produced by one labelling. Thus, a labelling of a species uniquely defines a strand graph. Since a strand graph has a unique canonical labelling, the relation between a strand graph and the species it represents is exclusive.

## 4.3 PySB Model Generation

The graph processor generates the full reaction network for a DSD system, which provides information to simulate the system. RuleDSD uses the PySB framework for integration with the BioNetGen engine in order to simulate the generated reaction network. Since PySB is designed for rule-based modelling for biochemical systems, it easily enables a translation of the data structures the graph processor uses for modelling the DSD system to a PySB model. The PySB model is further prompted for a BioNetGen simulation.

There are three essential components in a PySB model, they are named *Monomer*, *Rule* and *Parameter* in PySB. We discuss the initialisations of these components for RuleDSD in the following paragraphs.

Monomers are the fundamental elements in the system whose behaviours form reactions in the system. In other words, monomers in a PySB model are indivisible.

Traditionally, monomers are the fundamental elements that constitute the species, because PySB modelling approach is designed for those systems in which all the possible species and reactions are not known before the initialisation of the models. In our case, despite that the species may be divisible in biological standards, RuleDSD does not need to track down the details of a reaction in the level of strands during simulation because it already has those details from its graph processor. Thus, RuleDSD uses species instead of strands as monomers in a PySB model.

Parameters are constant numerical values that represent biological constants. They are of needed for defining a Rule object. RuleDSD provides the reaction rates that a user defines as parameters in a PySB model.

Rules define reactions between the complexes, which matches the definition of reactions in the graph processor. Therefore, RuleDSD writes each reaction from the graph processor as an expression that encloses the reactants and products in PySB syntax and then provides it together with its reaction rate to a PySB model as a rule.

Hence, a PySB model that describes the reaction network for a DSD system is created with a complete list of possible species and a list of corresponding reactions, as well as the reaction rates. Note that the reaction rates are with respect to the rules in DSD modelling, i.e., reactions enabled by the same rule shall have the same reaction rate.

## 4.4 Simulation

The chemical kinetic theory for modelling reaction networks allows describing a set of reactions in terms of ordinary differential equations (ODEs). Therefore, one can simulate the chemical kinetic model by initialising a model with certain concentrations on the initial species and use numerical integration to solve the set of ODEs with respect to time. However, this approximation method is not useful if stochastic noises are concerned, i.e., when the concentrations are small for the species. In this situation, Gillespie's stochastic simulation algorithm (SSA) [14] is recommended for the simulation. BioNetGen (BNG) provides both simulation methods illustrated above [41].

PySB integrates with BNG, and thus allows simulating a PySB model using BNG's simulation engines. In addition, one can integrate the ODEs by a SciPy integrator supported by PySB.

RuleDSD provides two simulation methods for simulating DSD systems: BNG's SSA simulator and the SciPy ODE simulator. Both simulators are accessible to PySB models which are obtained in the RuleDSD pipeline before simulation. The default simulator for RuleDSD is BNG's SSA simulator as the DSD systems usually develop

large reaction networks.

## 4.5 Visualisation

Simulation results are also retrieved by the PySB framework in RuleDSD. The outputs of RuleDSD for a DSD system includes the plot providing the simulation results and the textual description of the reaction network given by the graph processor.

The textual description of a reaction network has the format illustrated in Figure 18. The first segment named "Species" shows all possible species in the reaction networks, each species is assigned an index and is expressed in a format similar to its labelling (only the separation mark '|' for strands in the labelling is neglected because a separation is indicated by a line break, this format is also applied for the input species). The second segment named "Reactions" provides all the possible reactions, each line presents a reaction, starting with its reaction type, then a readable formula of the reaction using the index of species defined in the first segment, and ends with a reaction rate. The third segment demonstrates an incidence matrix based on the reaction network, its row denotes species and its column denotes the edge from one species to another (0s denote no edges, 1s denote out-going edges and -1s denote incoming edges). We further discuss the visualisation of results in Section 5.

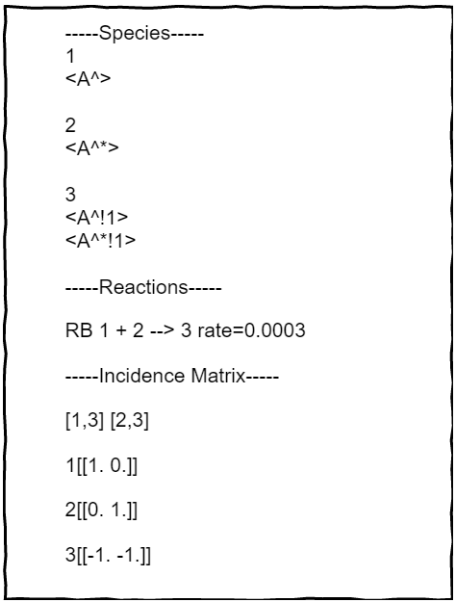



Figure 18: The DSDPy generated textual description of a reaction network.

# 5 Results

In this section, we present DSDPy, a software package implementing the RuleDSD pipeline. We provide user instructions for DSDPy as well as short discussions of its functions. We also present modelling of two DSD systems using DSDPy.

## 5.1 DSDPy

The DSDPy tool simplifies modelling and simulation of DSD systems. DSDPy is an open-source software that can be found at https://github.com/ashleylst/DSDPy.

There are two ways to use DSDPy. The traditional way is through the programming interface provided by Python. Recently, an interactive graphical user interface for DSDPy becomes available. The tool can also be used by an interactive graphical user interface (GUI). Full documentation for usage can be found at: https://dsdpy.readthedocs.io/en/latest/tutorial.html.

Here we provide a walk-through of its usage for generating and simulating the reaction network of a simple DSD system. Suppose the DSD system has two single-stranded DNA molecules as initial species, the first one has domains $t$ (a toehold domain) and $a$ (a non-toehold domain), and the second one has complementary domains of $t$ and $a$. We write the species in expressions $\langle t\hat{}\, a\rangle$ and $\langle a^*\ t\hat{}^*\rangle$.

A description of the DSD system is read from an existing file or entered manually, as shown on the top left of Figure 19. The description has three compulsory segments, each separated by two hyphens. The first segment includes the initial species, each separated by two forward slashes and is represented in a similar labelling form as discussed in Section 4.5. The second segment provides further details of the initial species, each line describes the name and then the concentration of a species. The order of lines in the second segment strictly follows the order of species introduced in the first segment. In our example, we name the first species `ss1` and the second `ss2`, both have initial population counts of 100. The third segment defines the reaction rates for each rule in the DSD system. DSDPy allows an additional fourth segment for simulation parameters. We set the simulation time to 5 seconds and the time step to 5 for our example.

During the reaction network generation, users are able to intervene by pausing, resuming and stopping the enumeration of all possible reactions. This interaction enables users to pre-examine the reactions and species in a DSD system, which is especially beneficial when the DSD system has a very large reaction network.

After a partial or full generation process is completed for the DSD system, a reaction network is formed and will be demonstrated as a textual description including the lists of species and reactions as shown in the text browser on the top right in Figure

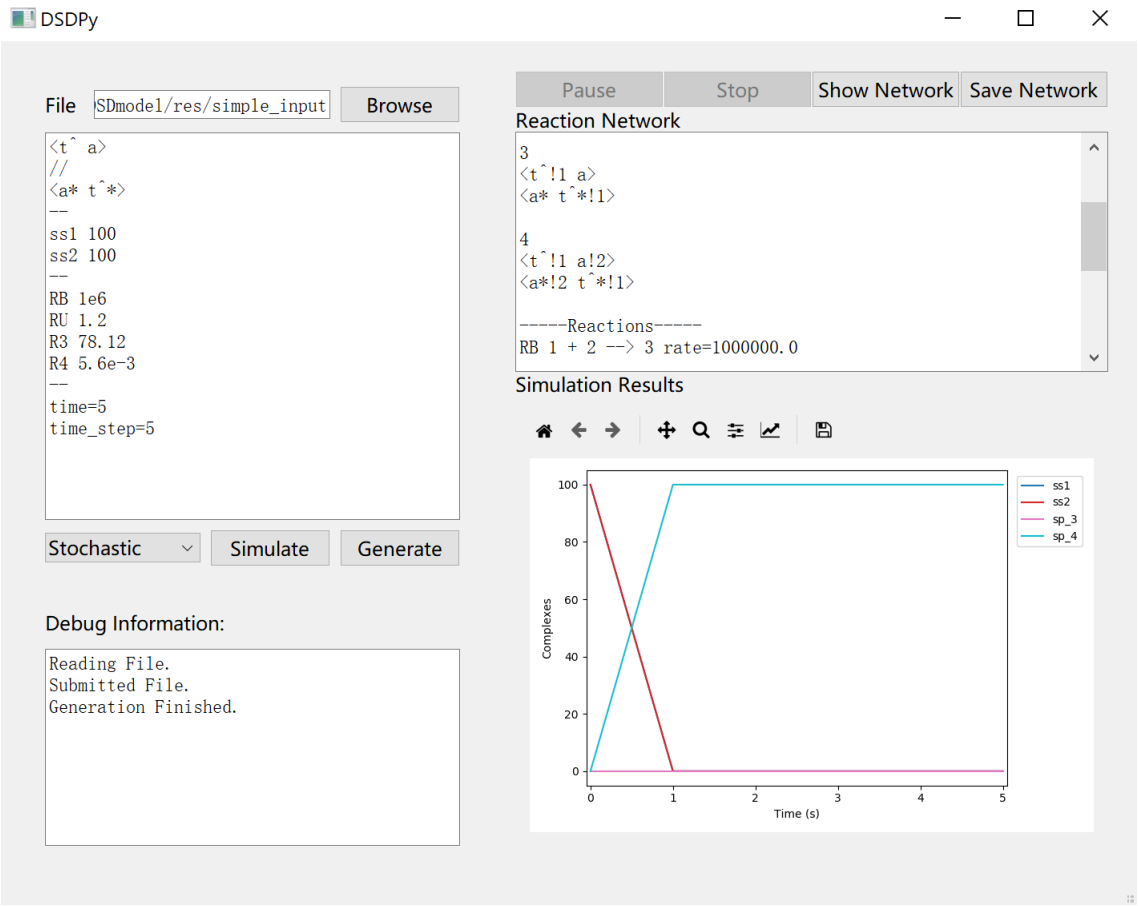



Figure 19: The DSDPy graphical user interface.

19. Users can simulate this reaction network by using either a stochastic or a deterministic simulation, and then the simulation results are plotted as a lineplot figure.

DSDPy has two highlighting features as a modelling tool for DSD systems. The first being the desirable usability it provides. The task of modelling a DSD system involves generating and simulating its reaction network, which is achieved by DSDPy with a minimal setup of DSD system description as input. Although there are certain rules a user should follow to provide the input, these rules are straightforward and do not require any programming skills. Since DSDPy has an interactive interface, it opens up the possibility for a controllable generation and simulation of a DSD reaction network.

The other major advantage of DSDPy is underlying programmability. DSDPy was originally designed as a Python package that serves as a backend engine for modelling DSD systems. Therefore, DSDPy implements the object-oriented programming (OOP) concepts to address the reusability and flexibility of the code. It implies that the modules and functions DSDPy contains are arranged carefully so that changes can be easily made. Nevertheless, one can use DSDPy as a software package and program with it conveniently.

The original design of DSDPy as a backend engine causes a problem with its design of input. Currently, the input for DSDPy is compact, thus it can be puzzling to those who are not familiar with the representations. Same can be said for the reaction network it produces as output. Although only a short explanation is needed, DSDPy has a potential to be integrated with a more user-friendly interface.

As any other rule-based modelling tool, DSDPy faces the challenge of combinatorial complexity [19], i.e., the number of generated species might increase exponentially after each iteration and lead to an explosion of species before the enumeration ends. Since DSDPy chooses to generate the full reaction network before simulation, there is a chance that the generation process enters a dead loop due to combinatorial complexity. There are mainly two ways to address this problem, one being that imposing restrictions on the reactions so that the reactions that could cause an explosion of species (e.g., repeated binding reactions) are cut off, the other being that adopting the simulation on-the-fly approach so that reactions with no available reactants are not explored. DSDPy uses a threshold to limit the number of iterations that can happen in the generation process, this nevertheless prevents an explosion of species, but is at the cost of a complete reaction network. The preferable solution for DSDPy might be using the simulation on-the-fly approach, though it would require DSDPy to directly deal with simulation methods.

## 5.2 Modelling a DSD System of Three-way Initiated Four-Way Branch Migration

This DSD system was introduced to demonstrate a new modelling feature implemented in Visual DSD to address the problem of modelling the DSD systems with secondary structures, including branches and loops [30]. It was taken partly from [34].

Figure 20 shows a handmade DSD reaction network from [30], such network can be successfully produced by the Visual DSD tool. This DSD system has two initial species and can generate up to six new species. Starting with a binding reaction, the resulting species is available for a three-way branch migration and then forms a Holliday junction and subsequently releases a single strand. The species with a Holliday junction is available for a four-way branch migration, and then an unbinding reaction further separates two double-stranded molecules from the species.

We provide the description of this DSD system as an input to DSDPy, with the same reaction rates and concentrations defined in [30]. We name the initial species `ss1` and `ss2` as shown in Figure 21.

DSDPy generates two more species (`sp_6` and `sp_9` in Figure 22) than what was reported in Figure 20. These two species are considered reachable in DSDPy because two different four-way branch migration reactions can be applied to species `sp_3`,

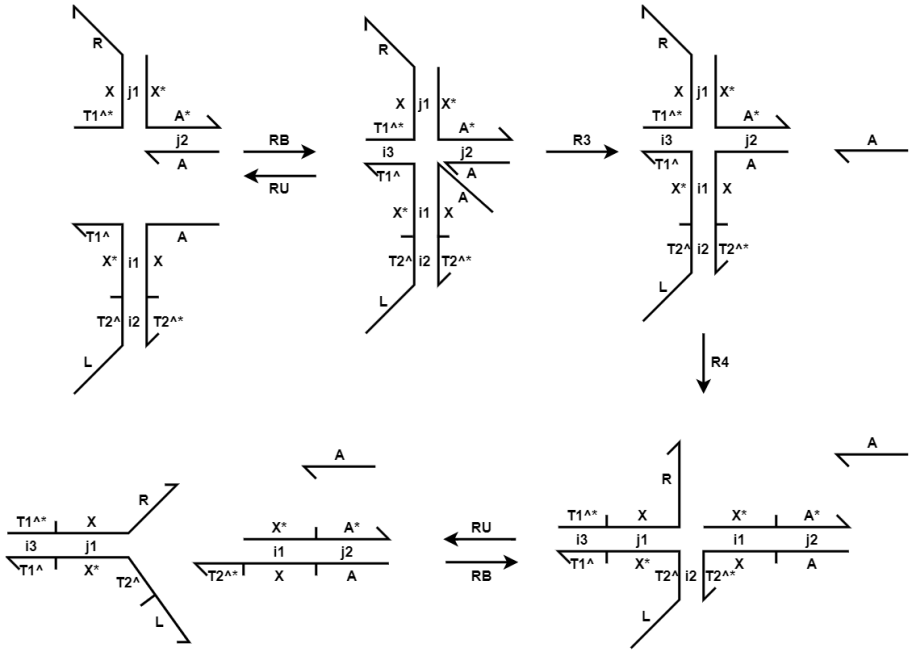



Figure 20: The DSD system illustrated in Figure 1 d in [30].

```
<L T2^!i2 X*!i1 T1^>
<A X!i1 T2^*!i2>
//
<T1^* X!j1 R>
<X*!j1 A*!j2>
<A!j2>
--
ss1 1000
ss2 1200
--
RB 0.0003
RU 0.113
R3 20
R4 20
```

Figure 21: Corresponding input describing the three-way initiated four-way branch migration example DSD system for DSDPy.

whereas only one is allowed by Visual DSD. This divergence exists because RuleDSD defines R4 as a pattern that matches not only species with closed junctions but also freely branched species (see Figure 16). The additional four-way branch migration reaction from species `sp_3` to `sp_6` further leads to species `sp_9` as species `sp_6` can initiate an unbinding reaction. Other than these two extra species and the reactions they are involved in, the DSDPy generated reaction network is identical to that generated by Visual DSD.

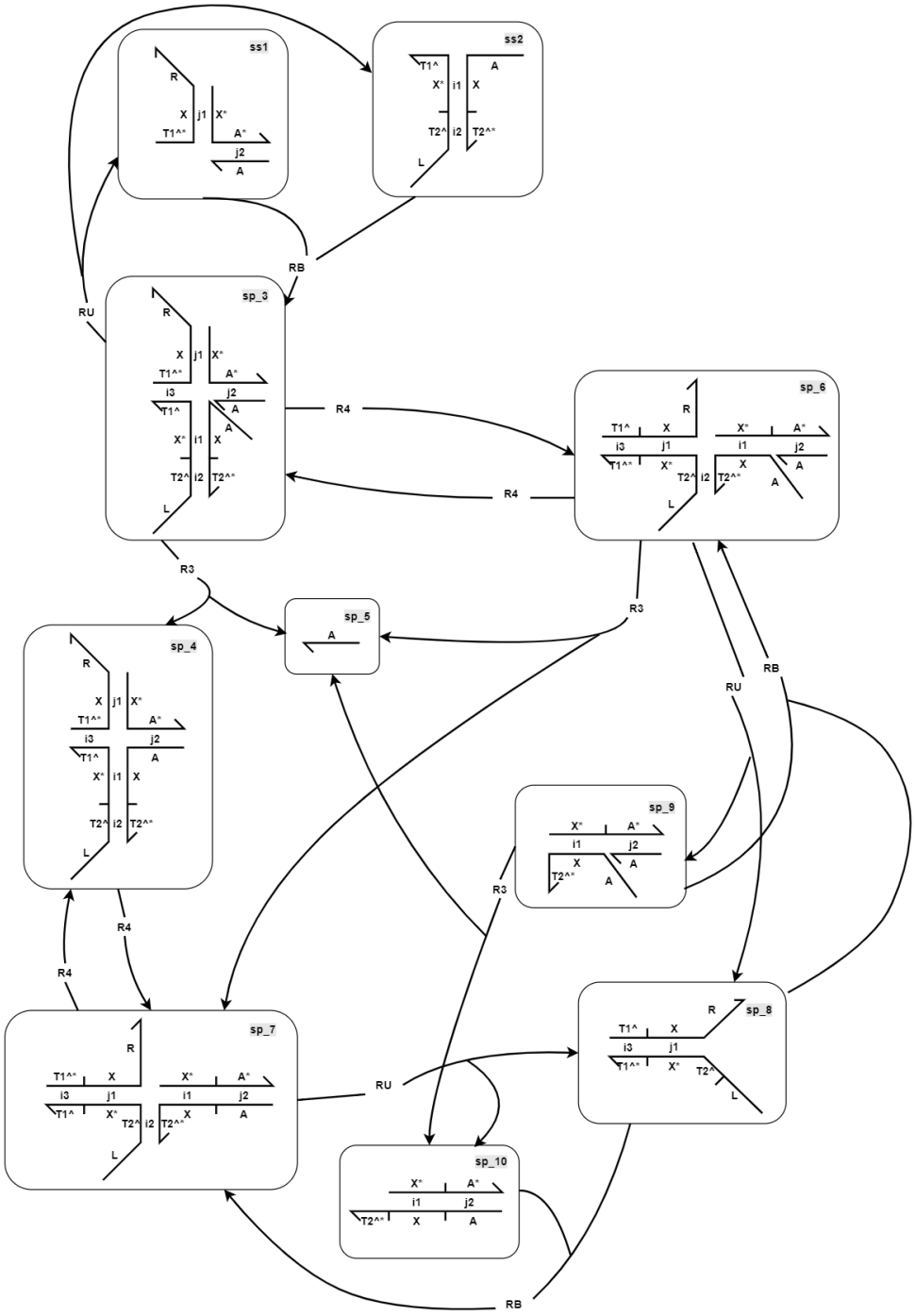



Figure 22: DSDPy generated reaction network of the three-way initiated four-way branch migration example. The arrows go from reactants to products, and corresponding reaction rates are marked for the reactions. Each box denotes a species with a species name in the grey box on the top right.

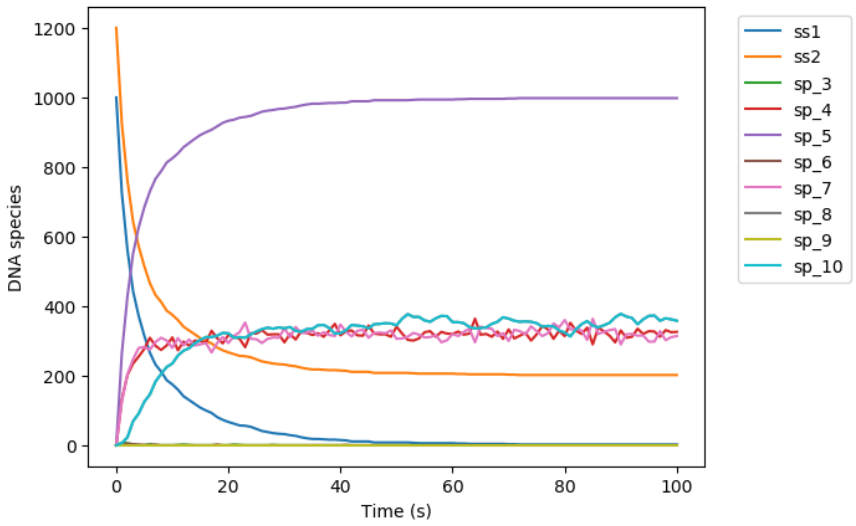



Figure 23: Results of a stochastic simulation of the three-way initiated four-way branch migration example. The legends refer to the species in Figure 22.

We have run a stochastic simulation on the DSDPy generated reaction network, the results are shown in Figure 23. We observe that the binding reaction of the two initial species (`ss1` and `ss2`) dramatically leads to the release of species `sp_5`, a single-stranded species $\langle A \rangle$. The products of three-way and four-way branch migration reactions are the other major elements in the system after the binding reaction happen (see species `sp_4` in red and `sp_7` in pink). Products from the unbinding reaction of species `sp_7` increase soon after the migration occurs (see species `sp_8` in grey and `sp_10` in light blue, these two lines coincide in the figure). Notice that there are very few species `sp_3`, `sp_6` and `sp_9` in the beginning of the simulation, and then they are transformed into other species.

## 5.3 Modelling a Single-layer Catalytic DSD System

This case study was reported by Kotani et al. as an example single-layer catalytic DSD (SCD) system using three-way branch migration to maximize catalysis and four-way branch migration to minimize leakage, i.e., waste species [22]. In the SCD system shown in Figure 24, Substrate S1 and Catalyst C1 bind together and initiate a three-way branch migration, producing Intermediate I1 and Product P1. I1 then reacts with Substrate S2 similarly and produces Intermediate I2 and C1. I2 can initiate a four-way branch migration to produce Product P2 and Product P3. In addition, Reporter R is designed with a quencher strand $\langle 3\hat{}^* \; d1^* \; d2^* \rangle$ and a dye strand $\langle d1 \; d2 \rangle$, resulting in suppressed fluorescence. R can react with P3 and produce Reporter waste Rw, releasing Dye strand D, which increases the fluorescence intensity for observation.

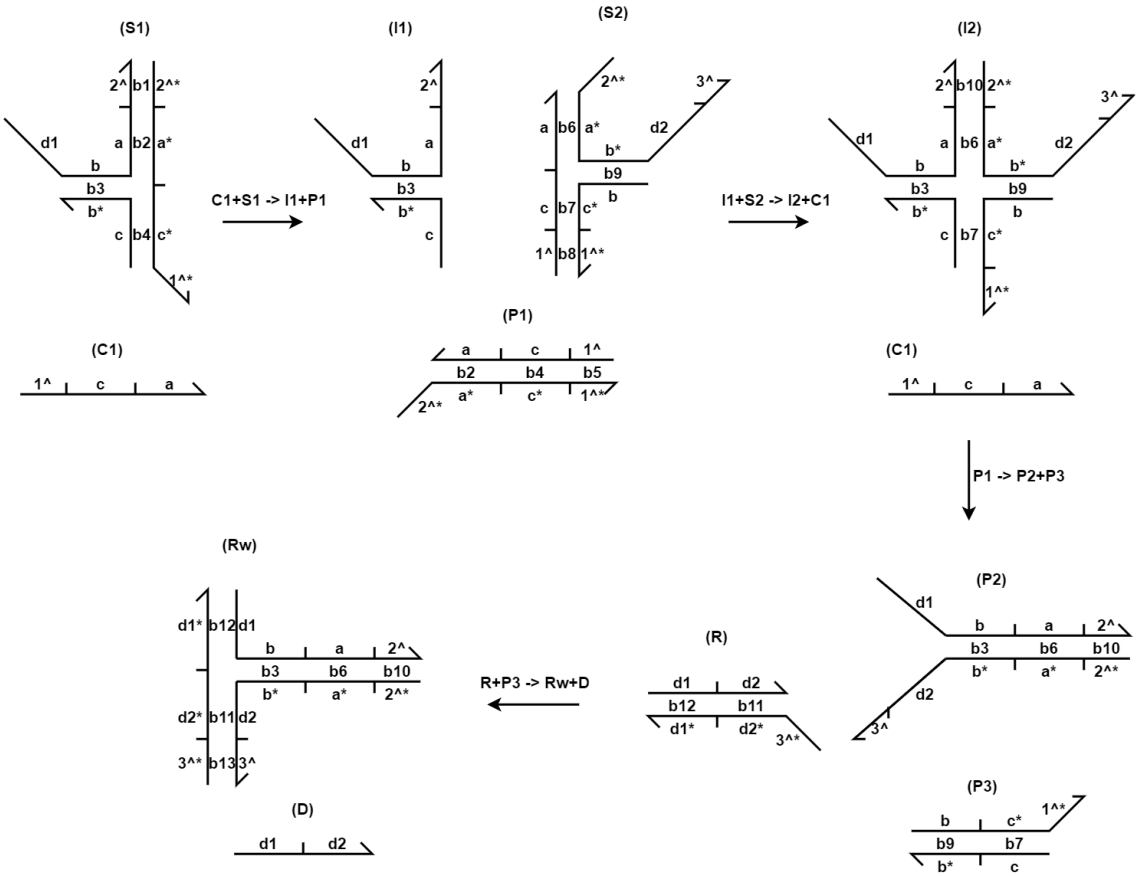



Figure 24: A handmade reaction network of the SCD system from [22]. The species names are shown on the top of the schematic drawings.

We provide initial species S1, S2, C1 and R to DSDPy, then it generates hundreds of species due to potential chain reactions. Therefore, we set an enumeration threshold to the generation process. With a threshold value of 6, this catalytic system generates 26 species in addition to the initial species, the reaction network concerning the species listed in [22] is shown in Figure 25. Notably, the release of P2 (`sp_14`) can happen during a four-way branch migration at the same stage when I2 (`sp_16`) is produced. Full details on the reaction network can be found in Appendix A.2. The resulting reaction network shows that DSDPy not only ensures the correctness by reproducing the handmade network from [22], but also explores unexpected products that are not in the list of possible species of the handmade network.

We also run a stochastic simulation of the resulting reaction network (the corresponding input file can be found in Appendix A.1), the plot concerning the possible species listed in [22] is shown in Figure 26. We observe that the population counts of S1 and C1 drop instantly when the simulation starts due to the binding reaction between them. Similarly, the populations of S2 and R decrease at the same rate because a binding reaction is also available for them. Species `sp_8` (I1) and `sp_9`

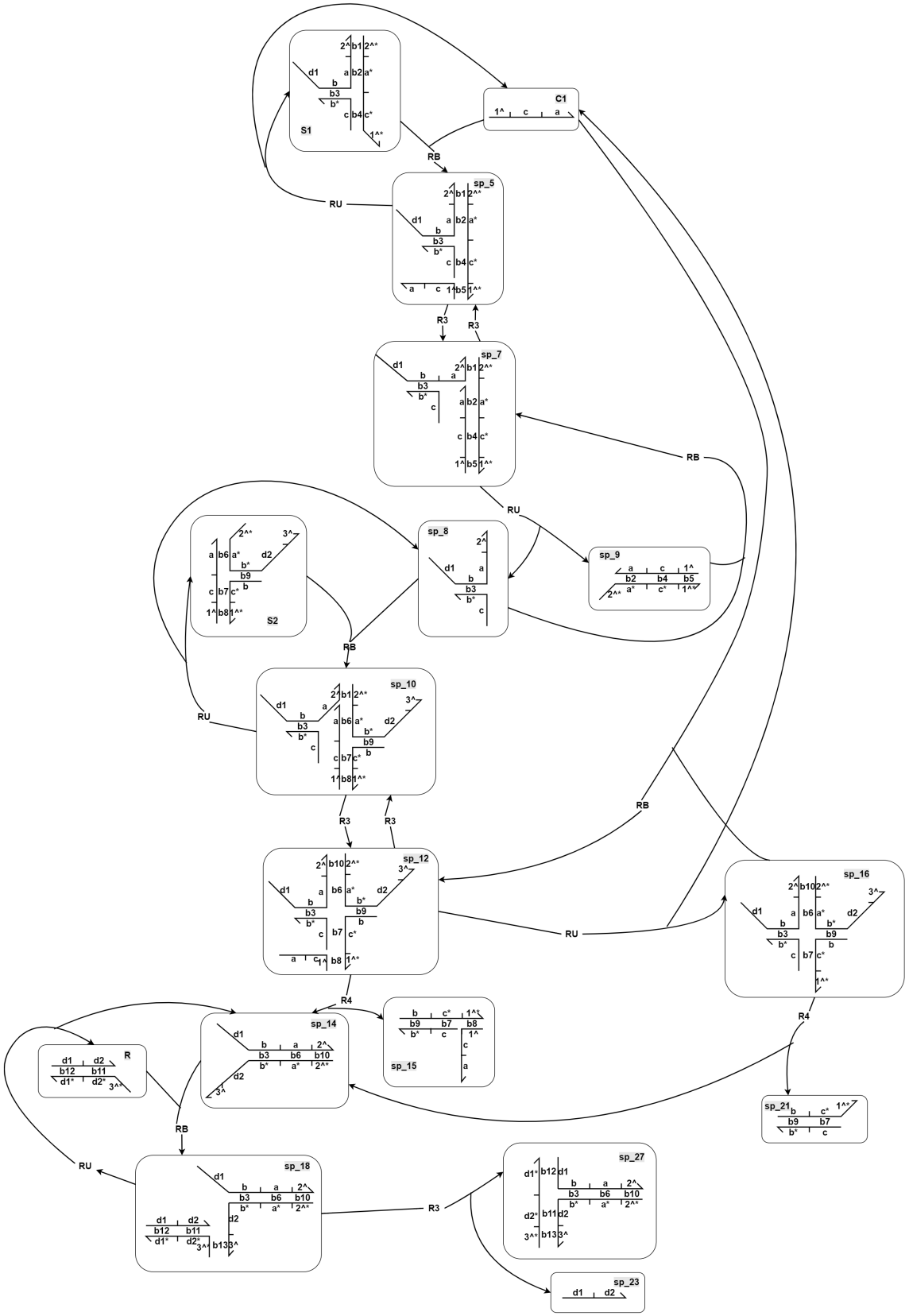



Figure 25: A truncated reaction network of the SCD system generated by DSDPy.

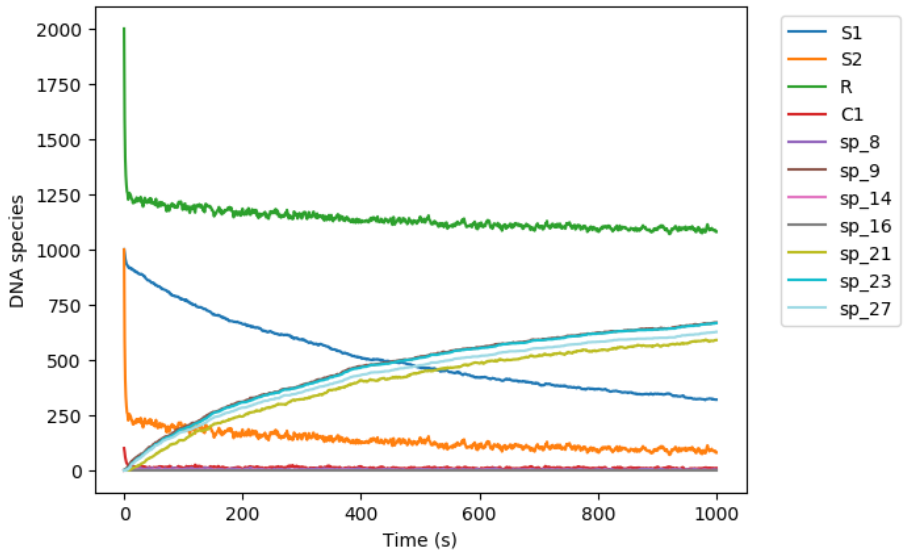



Figure 26: Results (partial) of a stochastic simulation of the single-layer catalytic system. The legends refer to the species in Figure 25. The full results are shown in Appendix A.3.

(P1) emerge at the same time, but `sp_8` slowly reacts with S1, whereas `sp_9` remains being produced. Although species `sp_14` (P2) and `sp_21` (P3) are the products of the same four-way branch migration initiated by `sp_16` (I2), `sp_14` can participate in other reactions as a reactant, which implies that there are significantly more `sp_21` than `sp_14` in the system. Species `sp_14` reacts with R and produce `sp_27` (Rw) and `sp_23` (D).

# 6 Discussion

## 6.1 Conclusion

In this thesis, we studied the DNA strand displacement mechanism and presented a rule-based modelling approach to generate and simulate reaction networks of DSD systems. The design of such modelling tool is presented as a software pipeline RuleDSD and implemented as a Python package DSDPy.

We provide the workflow and concepts of RuleDSD in Section 3. The central idea of RuleDSD is to represent DNA molecules, which we term species, as *Strand Graphs* and *Bond Graphs*. These graphical structures enable RuleDSD to detect underlying particular patterns so that it can match these patterns with the corresponding DSD reaction types. Therefore, a full reaction network is generated after all the patterns on all possible species have been explored. These patterns are termed rules in the rule-based modelling approach. Section 4.1 introduces the algorithms used in RuleDSD to define the rules for DSD systems. We also highlight the algorithm for mapping a newly generated species with the species at hand in Section 4.2, as it plays an important part in the generation of reaction networks.

After the reaction network generation is finished, RuleDSD transfers the reaction network information to a PySB model, which is simulated using either a deterministic or stochastic simulation as required by a user. Section 4.3 to 4.5 specify the implementation details of the simulation process.

We presented DSDPy as a software tool in Section 5. We show that DSDPy has the capability to model secondary structures such as loops and multiple branches of DNA molecules and identify all possible reactions along with all possible species given the initial species of a DSD system. We studied several DSD systems from existing literature. In general, DSDPy based modelling and simulation comply with the results from other state-of-the-art DSD modelling tools, such as Visual DSD. However, in some cases such as the DSD systems presented in Section 5.2 and 5.3, DSDPy generates additional species and more detailed reaction networks.

Altogether, DSDPy simplifies the modelling of DSD systems. Rule-based DSD modelling approach enables the designs of human-readable languages for modelling purposes. DSDPy further frees users from programming language knowledge and allows them to work with fewer professional insights. For more experienced users, DSDPy also offers opportunities to be tuned for individual and specific use as it is a lightweight open-source Python package.

## 6.2 Future Work

The RuleDSD model currently covers only four types of basic DSD reactions: binding, unbinding, three-way branch migration and four-way branch migration. Although a variety of DSD systems from literature can be modelled using the present RuleDSD, there are other relevant detailed reaction types in DSD systems, such as hairpin closing, bulge closing and multiloop closing with respect to binding illustrated in [16]. In future, these additional reaction types can be included in RuleDSD by adding corresponding rules.

As discussed in Section 5.1, DSDPy currently does not address the problem of combinatorial complexity because it uses a generate-then-simulate approach for modelling DSD systems. However, a very small DSD system can lead to an unbounded reaction network. For example, a system with strands $\langle a\hat{}\ b\hat{}\rangle$ and $\langle a\hat{}^*\ b\hat{}^*\rangle$ results in an evergrowing reaction network as binding reactions happen endlessly. The evergrowing reaction network would force DSDPy to stay in the generation process unless a user pauses or terminates it. This may be an inefficient use of computational resources. A preferred solution to this problem would be adopting a simulation-on-the-fly approach introduced by [11]. Such approach can not be implemented directly in the present DSDPy because an external simulator is used. It would require a combined implementation of generation and simulation in DSDPy.

In this thesis, we have focused on modelling domain-level DSD systems, but the sequence-level information is not to be neglected because only that can fully explain the biophysical reactions. As what was pointed out by [30], a strand graph may define any arbitrary secondary structures without considering if they are biophysically plausible. For instance, a hairpin loop with bonded domains inside would not be feasible unless the combined length of the bonded domains is longer than the persistent length (the length which quantifies the bending stiffness) of the double-stranded DNA molecules. DSDPy addresses the biophysical plausibility of strand graphs by doing various checks before applying rules, but it does not yet consider DSD systems at sequence-level. Namely, DSDPy decides the validity of a reaction without acknowledging the possibility that such decision may be incorrect when the lengths of domains are considered. To include sequence-level information in DSDPy, one can use sequence design software such as NUPACK [54] to generate sequences of nucleotides for each domain of a given DSD system. Additional checks can be implemented to ensure complementarity of domains in the strand graphs. However, detailed modelling at sequence-level would require rigorous verification and analysis to check up to what extent the designs at domain-level and at sequence-level conform.

# References


[1] Berleant, J., Berlind, C., Badelt, S., Dannenberg, F., Schaeffer, J., and Winfree, E. Automated sequence-level analysis of kinetics and thermodynamics for domain-level DNA strand-displacement systems. *Journal of the Royal Society Interface 15*, 149 (2018), 20180107.

[2] Bollobás, B. *Modern Graph Theory*, vol. 184. Springer Science & Business Media, 2013.

[3] Bui, H., Miao, V., Garg, S., Mokhtar, R., Song, T., and Reif, J. Design and analysis of localized DNA hybridization chain reactions. *Small 13*, 12 (2017), 1602983.

[4] Chen, J., and Seeman, N. C. Synthesis from DNA of a molecule with the connectivity of a cube. *Nature 350*, 6319 (1991), 631–633.

[5] Chylek, L. A., Harris, L. A., Faeder, J. R., and Hlavacek, W. S. Modeling for (physical) biologists: an introduction to the rule-based approach. *Physical Biology 12*, 4 (2015), 045007.

[6] Chylek, L. A., Harris, L. A., Tung, C.-S., Faeder, J. R., Lopez, C. F., and Hlavacek, W. S. Rule-based modeling: a computational approach for studying biomolecular site dynamics in cell signaling systems. *Wiley Interdisciplinary Reviews: Systems Biology and Medicine 6*, 1 (2014), 13–36.

[7] Dabby, N. L. *Synthetic Molecular Machines for Active Self-assembly: Prototype Algorithms, Designs, and Experimental Study*. PhD thesis, California Institute of Technology, 2013.

[8] Danos, V., and Laneve, C. Formal molecular biology. *Theoretical Computer Science 325*, 1 (2004), 69–110.

[9] Dirks, R. M., Lin, M., Winfree, E., and Pierce, N. A. Paradigms for computational nucleic acid design. *Nucleic Acids Research 32*, 4 (2004), 1392–1403.

[10] Endy, D., and Brent, R. Modelling cellular behaviour. *Nature 409*, 6818 (2001), 391–395.

[11] Faeder, J. R., Blinov, M. L., Goldstein, B., and Hlavacek, W. S. Rule-based modeling of biochemical networks. *Complexity 10*, 4 (2005), 22–41.

[12] Faeder, J. R., Hlavacek, W. S., Reischl, I., Blinov, M. L., Metzger, H., Redondo, A., Wofsy, C., and Goldstein, B. Investigation of early events in fc$\varepsilon$ri-mediated signaling using a detailed mathematical model. *The Journal of Immunology 170*, 7 (2003), 3769–3781.

[13] Gautam, V., Long, S., and Orponen., P. RuleDSD: A rule-based modelling and simulation tool for DNA strand displacement systems. In *Proceedings of the 13th International Joint Conference on Biomedical Engineering Systems and Technologies - Volume 3 BIOINFORMATICS* (2020), SciTePress, pp. 158–167.

[14] Gillespie, D. T. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics 22*, 4 (1976), 403–434.

[15] Gong, H., Zuliani, P., Komuravelli, A., Faeder, J. R., and Clarke, E. M. Analysis and verification of the hmgb1 signaling pathway. In *BMC Bioinformatics* (2010), vol. 11(Suppl 7), Springer.

[16] Grun, C., Sarma, K., Wolfe, B., Shin, S. W., and Winfree, E. A domain-level DNA strand displacement reaction enumerator allowing arbitrary non-pseudoknotted secondary structures. *arXiv preprint arXiv:1505.03738* (2015).

[17] Grun, C., Werfel, J., Zhang, D. Y., and Yin, P. DyNAMiC Workbench: an integrated development environment for dynamic DNA nanotechnology. *Journal of the Royal Society Interface 12*, 111 (2015), 20150580.

[18] Harris, L. A., Hogg, J. S., Tapia, J.-J., Sekar, J. A., Gupta, S., Korsunsky, I., Arora, A., Barua, D., Sheehan, R. P., and Faeder, J. R. BioNetGen 2.2: advances in rule-based modeling. *Bioinformatics 32*, 21 (2016), 3366–3368.

[19] Hlavacek, W. S., Faeder, J. R., Blinov, M. L., Perelson, A. S., and Goldstein, B. The complexity of complexes in signal transduction. *Biotechnology and Bioengineering 84*, 7 (2003), 783–794.

[20] Hlavacek, W. S., Faeder, J. R., Blinov, M. L., Posner, R. G., Hucka, M., and Fontana, W. Rules for modeling signal-transduction systems. *Science Signaling 2006*, 344 (2006), re6.

[21] Holliday, R. A mechanism for gene conversion in fungi. *Genetics Research 5*, 2 (1964), 282–304.

[22] Kotani, S., and Hughes, W. L. Multi-arm junctions for dynamic DNA nanotechnology. *Journal of the American Chemical Society 139*, 18 (2017), 6363–6368.

[23] Lakin, M. R., Paulevé, L., and Phillips, A. Stochastic simulation of multiple process calculi for biology. *Theoretical Computer Science 431* (2012), 181–206.

[24] Lakin, M. R., Youssef, S., Cardelli, L., and Phillips, A. Abstractions for DNA circuit design. *Journal of The Royal Society Interface 9*, 68 (2012), 470–486.

[25] Lakin, M. R., Youssef, S., Polo, F., Emmott, S., and Phillips, A. Visual DSD: a design and analysis tool for DNA strand displacement systems. *Bioinformatics 27*, 22 (2011), 3211–3213.

[26] Lopez, C. F., Muhlich, J. L., Bachman, J. A., and Sorger, P. K. Programming biological models in Python using PySB. *Molecular Systems Biology 9*, 1 (2013).

[27] Nag, A., Faeder, J. R., and Goldstein, B. Shaping the response: the role of fc$\varepsilon$ri and syk expression levels in mast cell signalling. *IET Systems Biology 4*, 6 (2010), 334–347.

[28] Oishi, K., and Klavins, E. Biomolecular implementation of linear i/o systems. *IET Systems Biology 5*, 4 (2011), 252–260.

[29] Oury, N., Pedersen, M., and Petersen, R. Canonical labelling of site graphs. *arXiv preprint arXiv:1306.2405* (2013).

[30] Petersen, R. L., Lakin, M. R., and Phillips, A. A strand graph semantics for DNA-based computation. *Theoretical Computer Science 632* (2016), 43–73.

[31] Phillips, A., and Cardelli, L. A programming language for composable DNA circuits. *Journal of the Royal Society Interface 6* (2009), S419–S436.

[32] Pray, L. Discovery of DNA structure and function:Watson and Crick. *Nature Education 1*, 1 (2008), 100.

[33] Qian, L., and Winfree, E. Scaling up digital circuit computation with DNA strand displacement cascades. *Science 332*, 6034 (2011), 1196–1201.

[34] Qian, L., and Winfree, E. Parallel and scalable computation and spatial dynamics with DNA-based chemical reaction networks on a surface. In *International Workshop on DNA-Based Computers* (2014), Springer, pp. 114–131.

[35] Richmond, T. J., and Davey, C. A. The structure of DNA in the nucleosome core. *Nature 423*, 6936 (2003), 145–150.

[36] Rothemund, P. W. Folding DNA to create nanoscale shapes and patterns. *Nature 440*, 7082 (2006), 297–302.

[37] Seelig, G., Soloveichik, D., Zhang, D. Y., and Winfree, E. Enzyme-free nucleic acid logic circuits. *Science 314*, 5805 (2006), 1585–1588.

[38] Seeman, N. C. Nucleic acid junctions and lattices. *Journal of Theoretical Biology 99*, 2 (1982), 237–247.

[39] Seeman, N. C. Structural DNA nanotechnology. In *NanoBiotechnology Protocols*. Springer, 2005, pp. 143–166.

[40] Seeman, N. C. Nanomaterials based on DNA. *Annual Review of Biochemistry 79* (2010), 65–87.

[41] Sekar, J. A. P., and Faeder, J. R. *Rule-Based Modeling of Signal Transduction: A Primer*. Humana Press, Totowa, NJ, 2012, pp. 139–218.

[42] Sherman, W. B., and Seeman, N. C. A precisely controlled DNA biped walking device. *Nano Letters 4*, 7 (2004), 1203–1207.

[43] Shih, W. M., Quispe, J. D., and Joyce, G. F. A 1.7-kilobase single-stranded DNA that folds into a nanoscale octahedron. *Nature 427*, 6975 (2004), 618–621.

[44] Shin, J.-S., and Pierce, N. A. A synthetic DNA walker for molecular transport. *Journal of the American Chemical Society 126*, 35 (2004), 10834–10835.

[45] Soloveichik, D., Seelig, G., and Winfree, E. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences 107*, 12 (2010), 5393–5398.

[46] Spaccasassi, C., Lakin, M. R., and Phillips, A. A logic programming language for computational nucleic acid devices. *ACS Synthetic Biology 8*, 7 (2018), 1530–1547.

[47] Voit, E. O. *Computational Analysis of Biochemical Systems: A Practical Guide for Biochemists and Molecular Biologists*. Cambridge University Press, 2000.

[48] Watson, J. D., Crick, F., et al. A Structure for Deoxyribose Nucleic Acid.

[49] Watson, J. D., and Crick, F. H. Genetical implications of the structure of deoxyribonucleic acid. *Nature 171*, 4361 (1953), 964–967.

[50] Winfree, E., Liu, F., Wenzler, L. A., and Seeman, N. C. Design and self-assembly of two-dimensional DNA crystals. *Nature 394*, 6693 (1998), 539–544.

[51] Yin, P., Choi, H. M., Calvert, C. R., and Pierce, N. A. Programming biomolecular self-assembly pathways. *Nature 451*, 7176 (2008), 318–322.

[52] Yurke, B., and Mills, A. P. Using DNA to power nanostructures. *Genetic Programming and Evolvable Machines 4*, 2 (2003), 111–122.

[53] Yurke, B., Turberfield, A. J., Mills, A. P., Simmel, F. C., and Neumann, J. L. A DNA-fuelled molecular machine made of DNA. *Nature 406*, 6796 (2000), 605–608.

[54] Zadeh, J. N., Steenberg, C. D., Bois, J. S., Wolfe, B. R., Pierce, M. B., Khan, A. R., Dirks, R. M., and Pierce, N. A. NUPACK: analysis and design of nucleic acid systems. *Journal of Computational Chemistry 32*, 1 (2011), 170–173.

[55] Zhang, D. Y., and Seelig, G. Dynamic DNA nanotechnology using strand-displacement reactions. *Nature Chemistry 3*, 2 (2011), 103.

[56] Zhang, D. Y., Turberfield, A. J., Yurke, B., and Winfree, E. Engineering entropy-driven reactions and networks catalyzed by DNA. *Science 318*, 5853 (2007), 1121–1125.

[57] Zhang, D. Y., and Winfree, E. Dynamic allosteric control of noncovalent DNA catalysis reactions. *Journal of the American Chemical Society 130*, 42 (2008), 13921–13926.

# A Inputs and Outputs of the SCD System Modelling

## A.1 Input

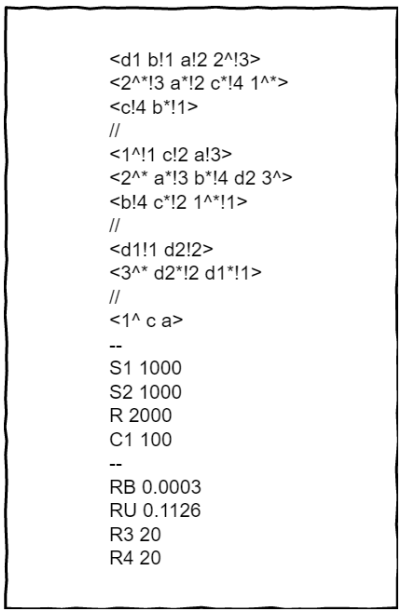



Figure A1: The SCD system input to DSDPy.

## A.2 Reaction Network of the SCD System

——Species——
1
<d1 b!1 a!2 2^!3>
<c!4 b*!1>
<2^*!3 a*!2 c*!4 1^*>

2
<1^!1 c!2 a!3>
<b!4 c*!2 1^*!1>
<2^* a*!3 b*!4 d2 3^>

3
<d1!1 d2!2>
<3^* d2*!2 d1*!1>

4
<1^ c a>

5
<d1 b!1 a!2 2^!3>
<c!4 b*!1>
<2^*!3 a*!2 c*!4 1^*!5>
<1^!5 c a>

6
<1^!1 c!2 a!3>
<b!4 c*!2 1^*!1>
<2^* a*!3 b*!4 d2 3^!5>
<3^*!5 d2*!6 d1*!7>
<d1!7 d2!6>

7
<d1 b!1 a 2^!2>
<c b*!1>
<2^*!2 a*!3 c*!4 1^*!5>
<1^!5 c!4 a!3>

8
<d1 b!1 a 2^>
<c b*!1>

9
<2^* a*!1 c*!2 1^*!3>
<1^!3 c!2 a!1>

10
<d1 b!1 a 2^!2>
<c b*!1>
<2^*!2 a*!3 b*!4 d2 3^>
<1^!5 c!6 a!3>
<b!4 c*!6 1^*!5>

11
<d1 b!1 a 2^!2>
<c b*!1>
<2^*!2 a*!3 b*!4 d2 3^!5>
<1^!6 c!7 a!3>
<b!4 c*!7 1^*!6>
<3^*!5 d2*!8 d1*!9>
<d1!9 d2!8>

12
<d1 b!1 a!2 2^!3>

<c!4 b*!1>
<2^*!3 a*!2 b*!5 d2 3^>
<b!5 c*!4 1^*!6>
<1^!6 c a>

13
<d1 b!1 a!2 2^!3>
<c!4 b*!1>
<2^*!3 a*!2 b*!5 d2 3^!6>
<b!5 c*!4 1^*!7>
<3^*!6 d2*!8 d1*!9>
<1^!7 c a>
<d1!9 d2!8>

14
<d1 b!1 a!2 2^!3>
<2^*!3 a*!2 b*!1 d2 3^>

15
<c!1 b*!2>
<b!2 c*!1 1^*!3>
<1^!3 c a>

16
<d1 b!1 a!2 2^!3>
<c!4 b*!1>
<2^*!3 a*!2 b*!5 d2 3^>
<b!5 c*!4 1^*>

17
<d1 b!1 a!2 2^!3>
<c!4 b*!1>
<2^*!3 a*!2 b*!5 d2!6 3^!7>
<b!5 c*!4 1^*!8>
<3^*!7 d2*!6 d1*!9>
<1^!8 c a>
<d1!9 d2>

18
<d1 b!1 a!2 2^!3>
<2^*!3 a*!2 b*!1 d2 3^!4>
<3^*!4 d2*!5 d1*!6>
<d1!6 d2!5>

19

<d1 b!1 a!2 2^!3>
<c!4 b*!1>
<2^*!3 a*!2 b*!5 d2 3^!6>
<b!5 c*!4 1^*>
<3^*!6 d2*!7 d1*!8>
<d1!8 d2!7>

20
<c b*!1>
<b!1 c*!2 1^*!3>
<1^!3 c!2 a>

21
<c!1 b*!2>
<b!2 c*!1 1^*>

22
<d1!1 b!2 a!3 2^!4>
<3^*!5 d2*!6 d1*!1>
<c!7 b*!2>
<2^*!4 a*!3 b*!8 d2!6 3^!5>
<b!8 c*!7 1^*!9>
<1^!9 c a>

23
<d1 d2>

24
<d1 b!1 a 2^!2>
<c b*!1>
<2^*!2 a*!3 b*!4 d2!5 3^!6>
<1^!7 c!8 a!3>
<b!4 c*!8 1^*!7>
<3^*!6 d2*!5 d1*!9>
<d1!9 d2>

25
<d1 b!1 a!2 2^!3>
<2^*!3 a*!2 b*!1 d2!4 3^!5>
<3^*!5 d2*!4 d1*!6>
<d1!6 d2>

26
<d1 b!1 a!2 2^!3>
<c!4 b*!1>

<2^*!3 a*!2 b*!5 d2!6 3^!7>
<b!5 c*!4 1^*>
<3^*!7 d2*!6 d1*!8>
<d1!8 d2>

27
<d1!1 b!2 a!3 2^!4>
<3^*!5 d2*!6 d1*!1>
<2^*!4 a*!3 b*!2 d2!6 3^!5>

28
<d1!1 b!2 a 2^!3>
<3^*!4 d2*!5 d1*!1>
<c b*!2>
<2^*!3 a*!6 b*!7 d2!5 3^!4>
<1^!8 c!9 a!6>
<b!7 c*!9 1^*!8>

29
<d1!1 b!2 a!3 2^!4>
<3^*!5 d2*!6 d1*!1>
<c!7 b*!2>
<2^*!4 a*!3 b*!8 d2!6 3^!5>
<b!8 c*!7 1^*>

30
<1^!1 c!2 a!3>
<b!4 c*!2 1^*!1>
<2^* a*!3 b*!4 d2!5 3^!6>
<3^*!6 d2*!5 d1*!7>
<d1!7 d2>

——Reactions——
RB 1 + 4 –> 5 rate=0.0003
RB 2 + 3 –> 6 rate=0.0003
R3 5 –> 7 rate=20.0
RU 5 –> 1 + 4 rate=0.1126
RU 6 –> 2 + 3 rate=0.1126
R3 7 –> 5 rate=20.0
RU 7 –> 8 + 9 rate=0.1126
RB 2 + 8 –> 10 rate=0.0003
RB 6 + 8 –> 11 rate=0.0003
R3 10 –> 12 rate=20.0
RU 10 –> 8 + 2 rate=0.1126
R3 11 –> 13 rate=20.0

RU 11 –> 8 + 6 rate=0.1126
RU 11 –> 10 + 3 rate=0.1126
RB 8 + 9 –> 7 rate=0.0003
RB 3 + 12 –> 13 rate=0.0003
R3 12 –> 10 rate=20.0
R4 12 –> 14 + 15 rate=20.0
RU 12 –> 16 + 4 rate=0.1126
R3 13 –> 11 rate=20.0
R3 13 –> 17 rate=20.0
R4 13 –> 18 + 15 rate=20.0
RU 13 –> 12 + 3 rate=0.1126
RU 13 –> 19 + 4 rate=0.1126
RB 3 + 14 –> 18 rate=0.0003
RB 3 + 16 –> 19 rate=0.0003
RB 4 + 16 –> 12 rate=0.0003
RB 4 + 19 –> 13 rate=0.0003
R3 15 –> 20 rate=20.0
RU 15 –> 21 + 4 rate=0.1126
R4 16 –> 14 + 21 rate=20.0
R3 17 –> 22 + 23 rate=20.0
R3 17 –> 24 rate=20.0
R3 17 –> 13 rate=20.0
R4 17 –> 25 + 15 rate=20.0
RU 17 –> 26 + 4 rate=0.1126
R3 18 –> 27 + 23 rate=20.0
RU 18 –> 14 + 3 rate=0.1126
R3 19 –> 26 rate=20.0
R4 19 –> 18 + 21 rate=20.0
RU 19 –> 16 + 3 rate=0.1126
RB 4 + 21 –> 15 rate=0.0003
RB 4 + 26 –> 17 rate=0.0003
R3 20 –> 15 rate=20.0
R3 22 –> 28 rate=20.0
R4 22 –> 27 + 15 rate=20.0
RU 22 –> 29 + 4 rate=0.1126
R3 24 –> 17 rate=20.0
R3 24 –> 11 rate=20.0
RU 24 –> 8 + 30 rate=0.1126
R3 25 –> 27 + 23 rate=20.0
R3 25 –> 18 rate=20.0
R3 26 –> 29 + 23 rate=20.0
R3 26 –> 19 rate=20.0
R4 26 –> 25 + 21 rate=20.0
RB 4 + 29 –> 22 rate=0.0003
RB 8 + 30 –> 24 rate=0.0003

——Incidence Matrix——
Omitted

## A.3 Simulation Results of the SCD System

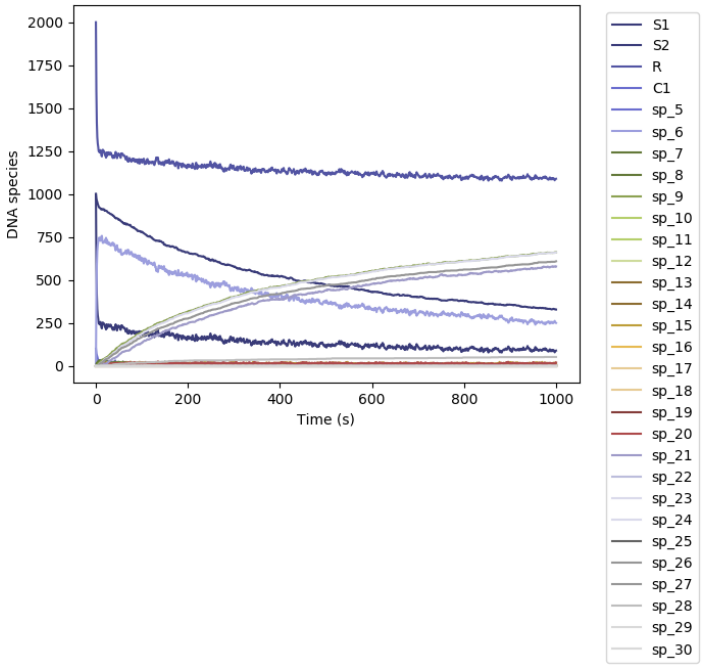



Figure A2: Simulation results of the the SCD System.